操作系统课程设计项目说明文档

MIT 操作系统实验 2022

|  |  |
| --- | --- |
| 作者姓名： | 史海博 |
| 学 号： | 2150759 |
| 指导教师： | 王冬青 |
| 学院专业： | 软件工程 |

同济大学

Tongji University

## 一 Lab: Xv6 and Unix Utilities

### 1.1 实验目的

1. 初步掌握 xv6 这一操作系统内核和用户。
2. 了解系统调用。
3. 了解父子进程，以及 fork()函数创建父子进程的方法。
4. 掌握素数筛，以及利用管道编写素数筛的方法。
5. 完成 find 和 xargs 程序的编写。

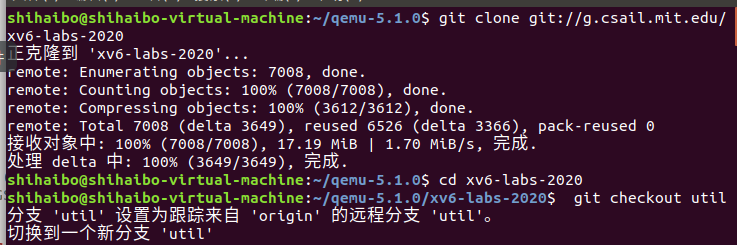
### 1.2 实验内容

1. 运行 xv6。
2. 为 xv6 实现 UNIX 程序 sleep：sleep 应该暂停一段用户所指定的时间。
3. 编写一个程序，使用 UNIX 系统调用在两个进程之间通过一对管道“乒乓” 一个字节，每个管道一个。
4. 使用管道编写并发版本的主筛。
5. 编写一个简单版本的 UNIX 查找程序：查找具有特定名称的目录树中的所有文件。
6. 编写一个简单版本的 UNIX xargs 程序：从标准输入中读取行并为每一行运行一个命令，将该行作为命令的参数提供。

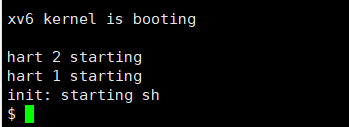
### 1.3 实验过程

#### 1.3.1运行 xv6

1. 切换到 util 分支：



1. 利用 make qemu 指令运行 xv6：

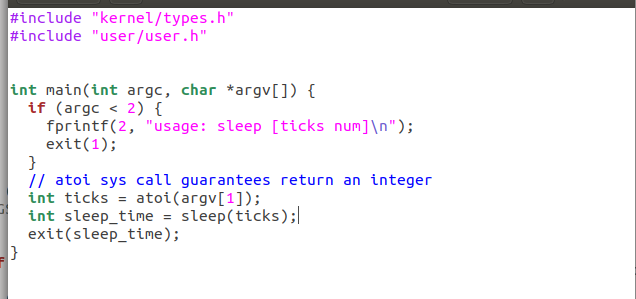


1. 使用 ls 命令查看文件：

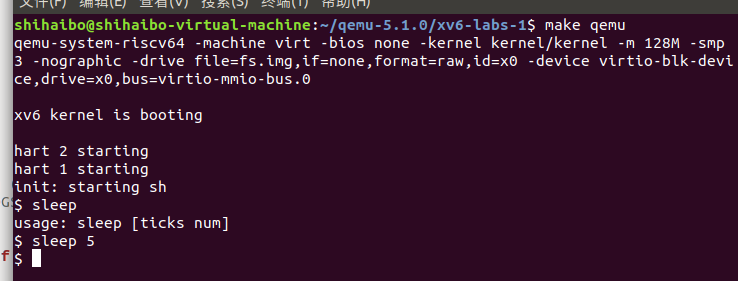


#### 1.3..2实现 sleep

1. 查看 sleep 系统调用函数，在sleep.c里修改sleep函数：



1. 最后，在 Makefile 文件中加上该方法即可。
2. 完成上述步骤后，我便可以在 xv6 中实现 sleep 方法：

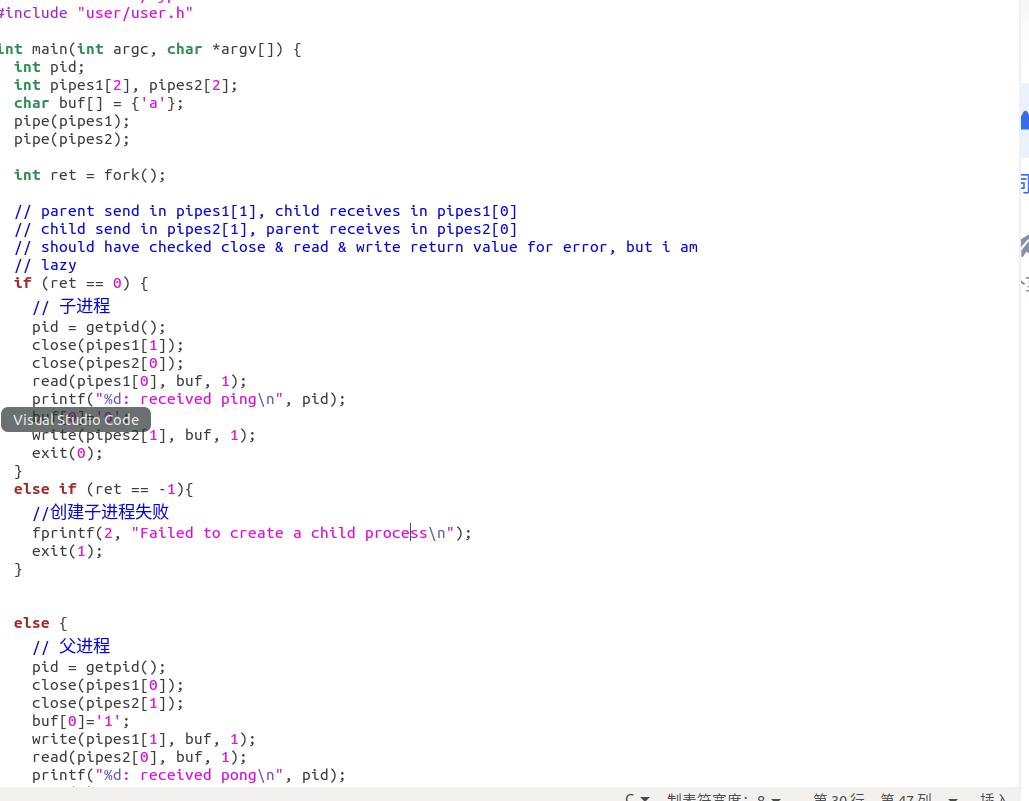


#### 1.3.3实现 pingpong

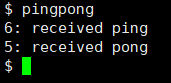
1. 根据题目所述，我应当先创建两个管道：

int fd1[2]; int fd2[2]; **pipe**(fd1); **pipe**(fd2);

1. 创建缓冲区，用于存储所需要传送的消息。同时，我借助 fork()创建子进程。需要注意到，我可以通过返回值来判断当前是父进程还是子进程：

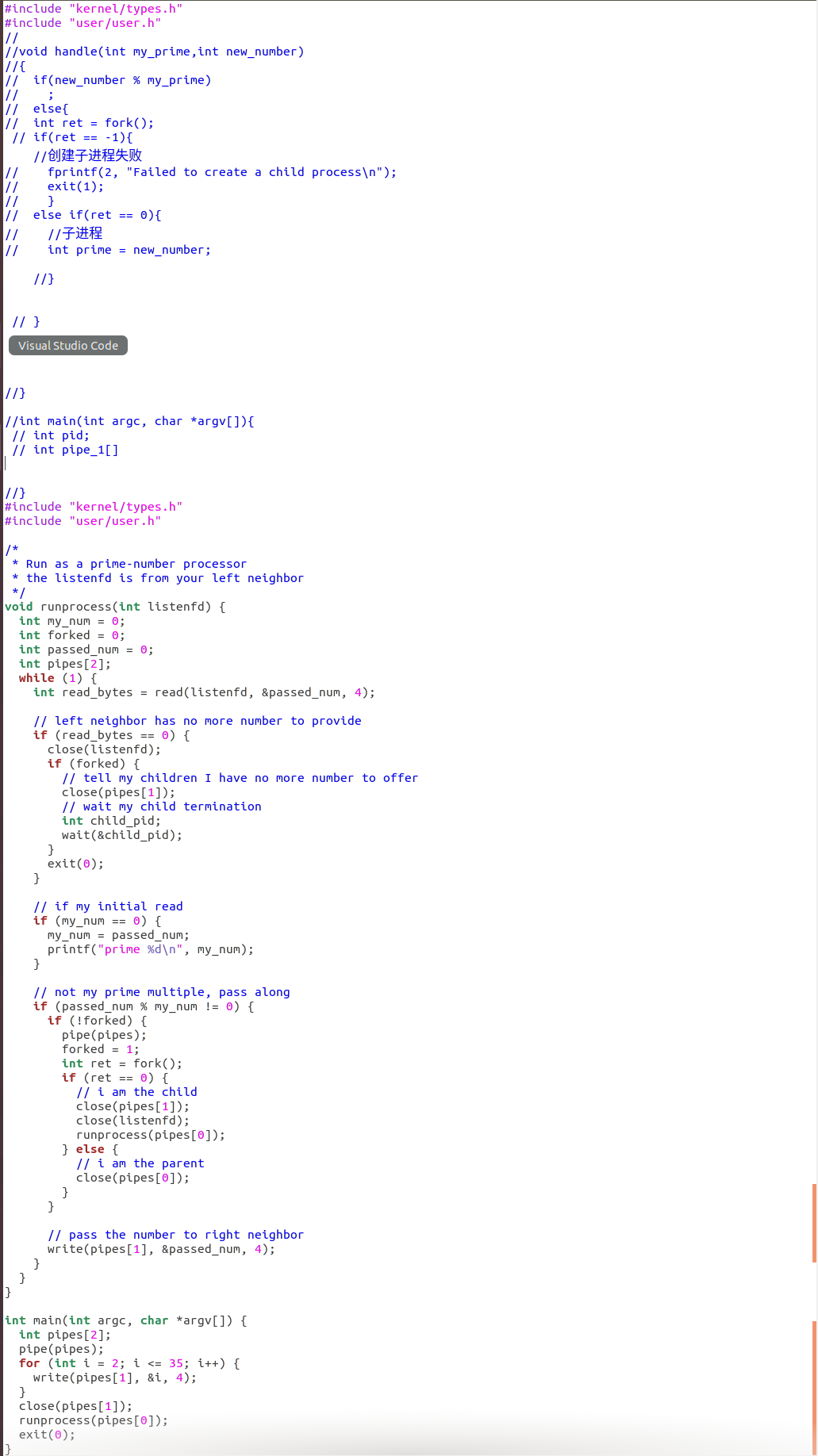


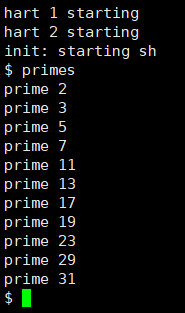
3.在 MakeFile 文件中添加该调用。完成后在 xv6 中测试：



#### 1.3.4实现 prime

1. 为了实现该功能，我在 main 函数中完成下述功能：

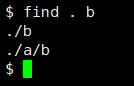


2.在 MakeFile 中添加 primes 之后，我便可以在 xv6 中运行 primes 命令，结果如下：

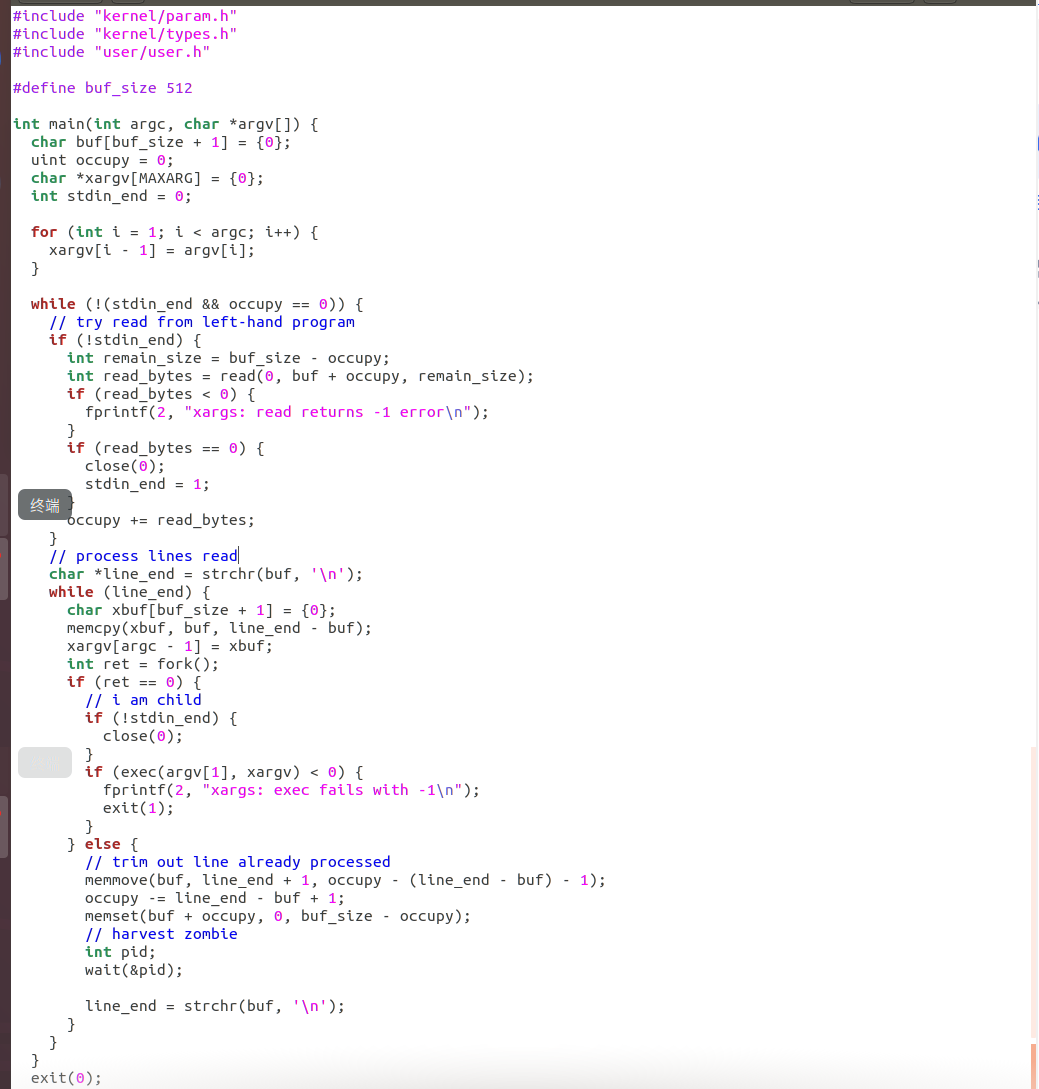
#### 1.3.5实现 find

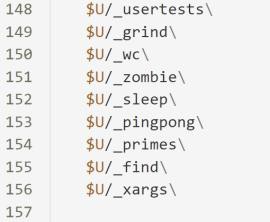
1.在user里分find.c里修改代码，根据一个初始路径和目标文件名, 要不断递归的扫描找到所有子目录前匹配的文件全路径.



2.完成上述代码后，在 MakeFile 文件中添加 find 方法。之后，便可以在 xv6 中使用 find

#### 1.3.6实现 xargs

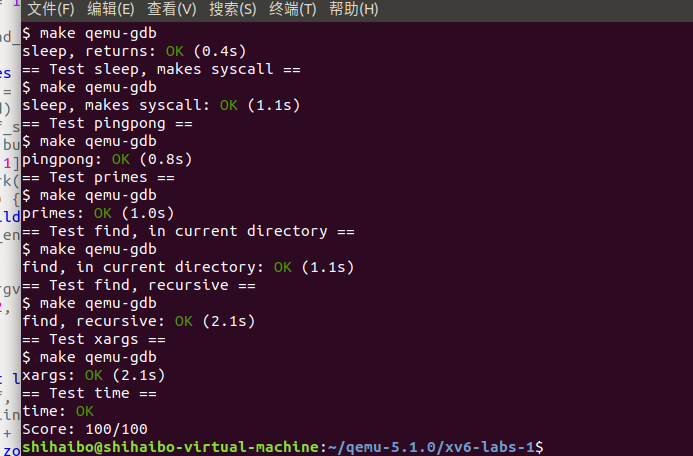
1. . 通过阅读本题题目的要求，可以发现需要解决的主要问题是对字符串进行处理。其中，| 之前的结果会在缓冲流中。因此，我需要通过读取缓冲流并且分解字符串（类似于 python 中的 split 函数）。
2. . 

3.添加到 MakeFile 文件中：

成功添加后，即可实现 xargs 功能：



#### 1.3.7综合测试



### 1.4 实验中的问题和解决方法

通过本实验，我初步了解了 xv6 这一操作系统内核，同时也了解了 qemu 模拟器的使用方法。

在完成实验的过程中，我遇到了比较多的困难，包括不知道管道如何创建、创建完成后如何关闭、素数筛到底是什么等等。但是通过和同学交流以及利用网络资源，我成功的解决了这些问题。

此外，我还初步了解到了系统调用接口，并且直到了操作系统内核和用户的一个区别。

### 1.5 实验心得

1. 理解Unix工具的概念和使用：实验中我学习了一些常用的Unix工具，如grep、ls、wc等，并深入理解它们的功能和使用方法。通过实际运行这些工具并观察其输出结果，我更好地理解了它们在文件处理、文本搜索和统计等方面的作用。

2. 学习xv6操作系统的文件系统和进程管理：实验要求我修改xv6内核代码，使得xv6能够正确运行Unix工具。这要求我深入了解xv6的文件系统和进程管理机制，以及它们与Unix工具的交互方式。通过阅读和修改xv6代码，我加深了对操作系统核心概念和机制的理解。

3. 编写Shell脚本和Makefile：实验中我需要编写Shell脚本和Makefile来测试和构建我的程序。这要求我熟悉Shell脚本的语法和用法，并学会使用Makefile来自动化编译和构建过程。通过编写这些脚本，我提高了自己的脚本编写能力和对构建工具的使用熟练度。

4. 系统调试和错误处理：在实验过程中，我遇到了一些bug和错误。通过使用GDB进行代码调试，并运行测试程序来验证我的修改，我学会了如何快速定位和解决问题，提高了我的调试能力。

5. 深入理解操作系统和工具的交互方式：实验要求我设计和实现一些Unix工具，使其能够与xv6操作系统正确地交互。这要求我深入思考和理解操作系统和工具之间的接口和交互方式，提高了我的系统设计和实现能力。

## 二 Lab: system calls

### 2.1 实验目的

1. 进一步了解系统调用。
2. 掌握添加系统调用的方法。
3. 理解系统调用的工作原理和 xv6 内核的工作过程。

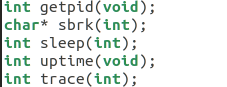
### 2.2 实验内容

1. 为 xv6 添加系统调用跟踪功能。
2. 添加系统调用 sysinfo，用于收集正在运行的系统的信息。

### 2.3 实验过程

#### 2.3.1 System call tracing

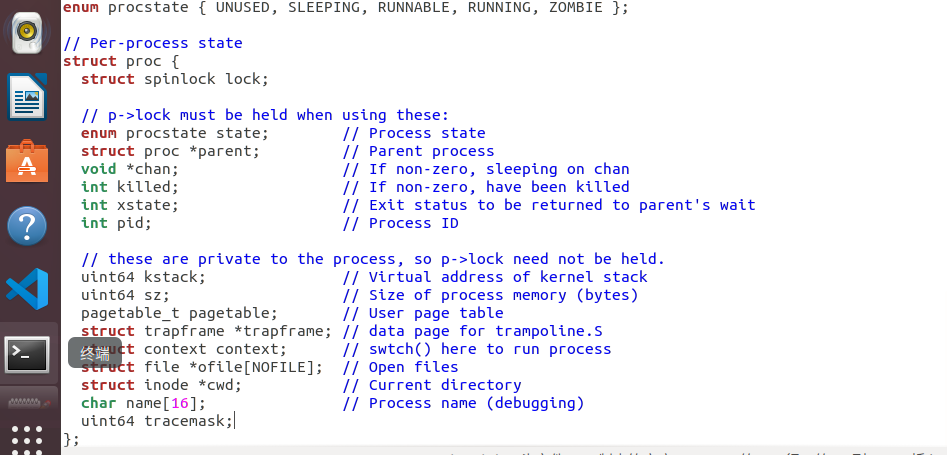
1. 首先我们在user/user.h里提供一个用户的接口. 只specify函数原型签名即可.



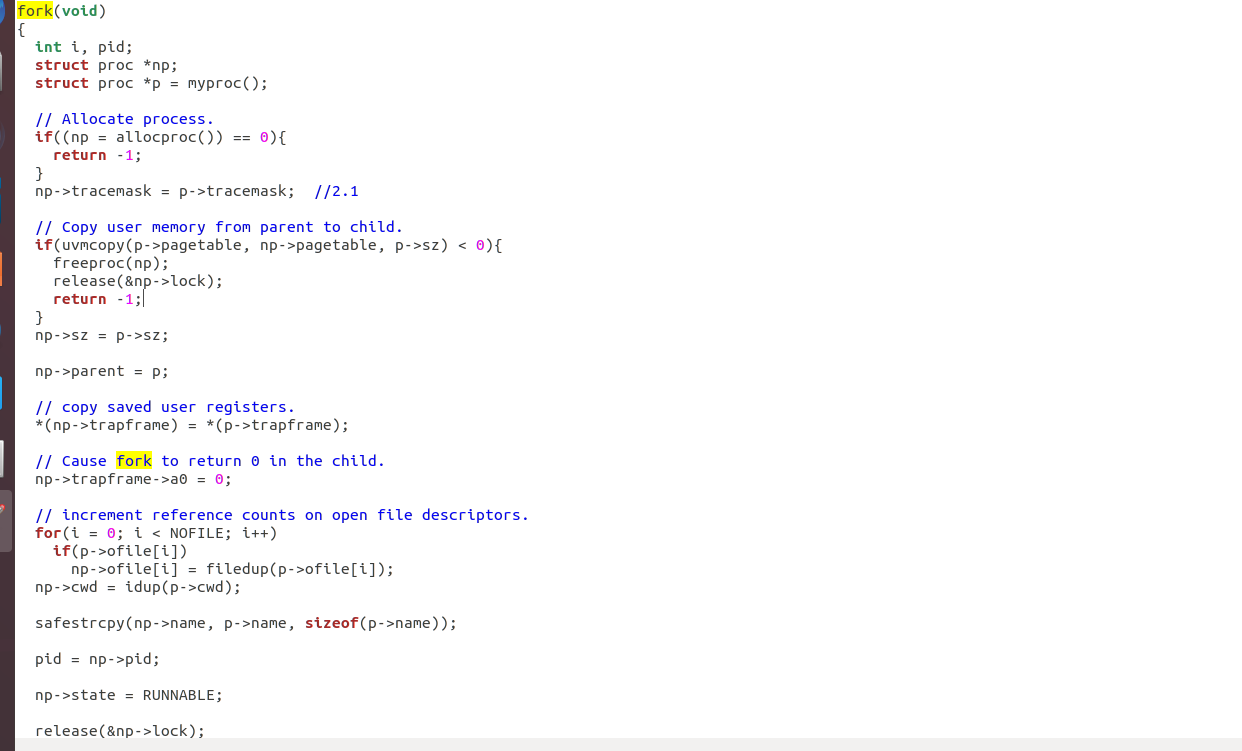
1. 接着在user/usys.pl里为这个函数签名加入一个entry.

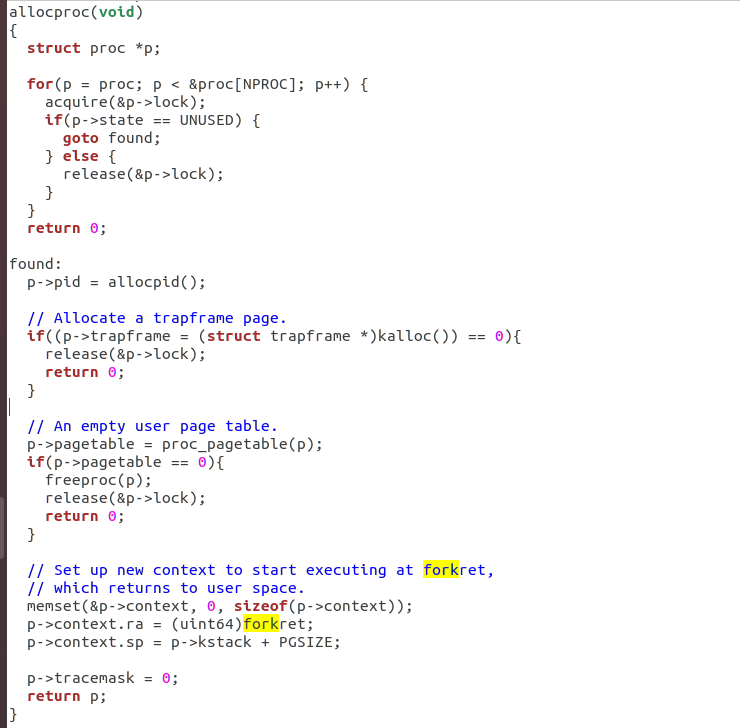


1. 为struct proc加上记忆需要trace的bit mask

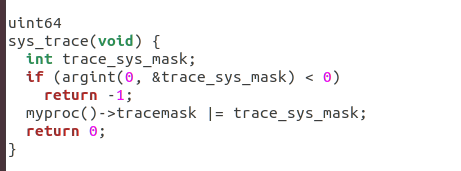


1. 在kernel/proc.c 里修改2个函数 allocproc(void)fork(void)



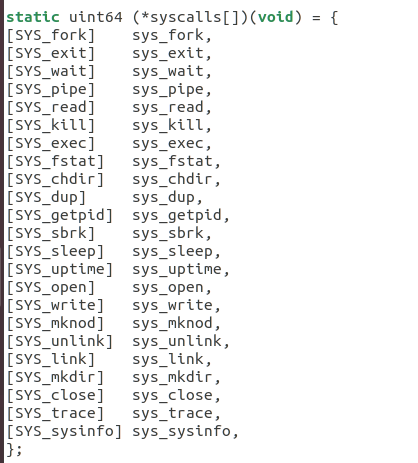


1. 实现内核态下的sys\_trace函数, 用argint函数从寄存器拿参数, 用myproc获得当前进程的一个指针

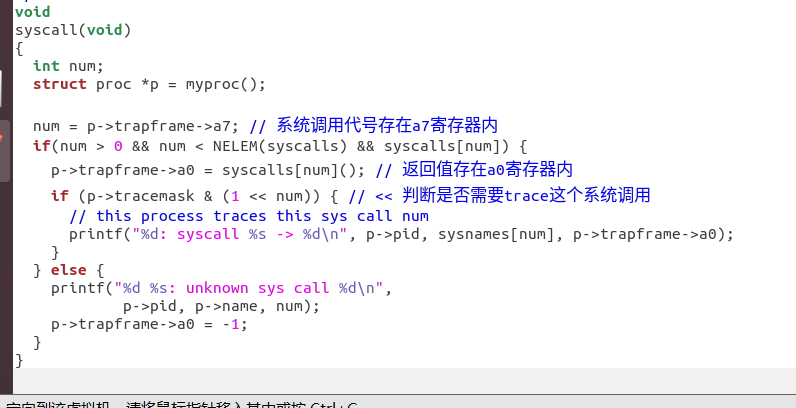


1. 为这个sys\_trace提供一个代号和函数指针的mapping. 并且顺手加上一个从代号到名字的mapping, 便于print

在syscall.h中加入宏常量

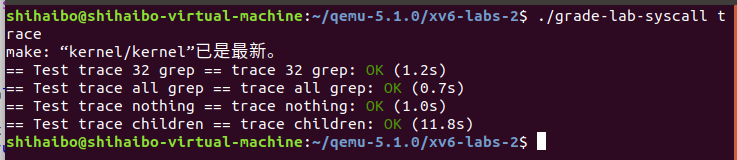


1. 最后修改通用入口syscall函数, 让它print出需要trace的系统调用，在syscall.c里加入函数



1. 在 MakeFile 文件中添加 trace 后，即可成功运行

指令：./grade-lab-syscall trace

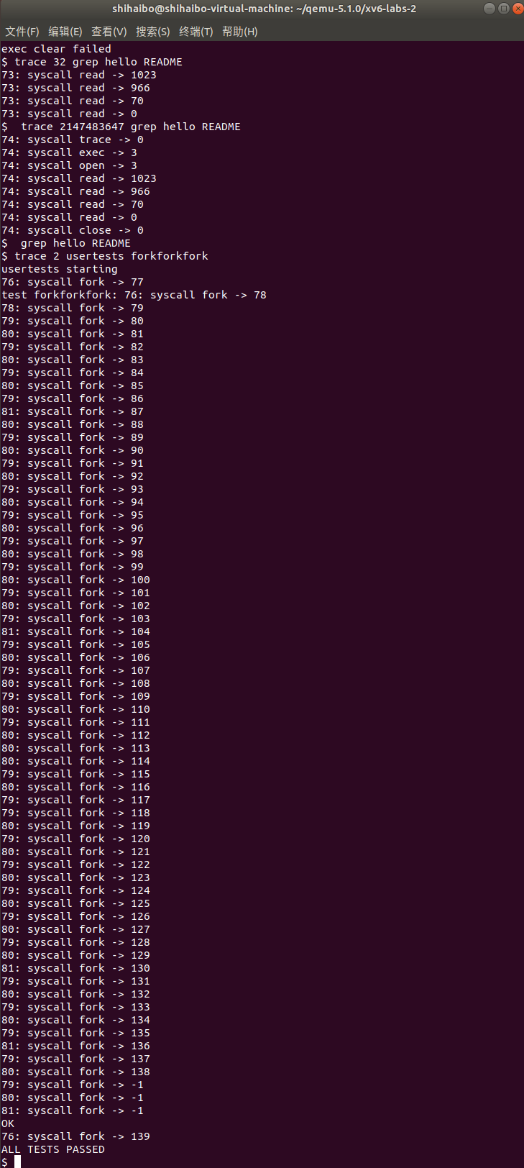


1. 测试结果

测试指令:

1. $ trace 32 grep hello README
2. trace 2147483647 grep hello README
3. grep hello README

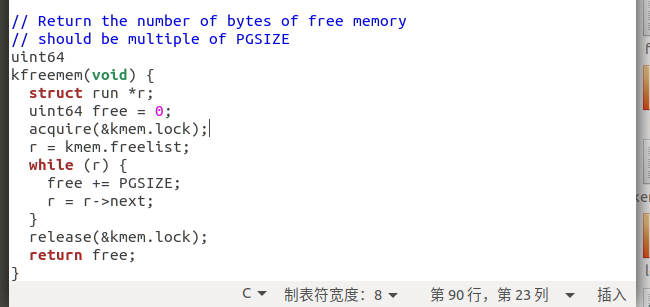
trace 2 usertests forkforkfork



#### 2.3.2 Sysinfo

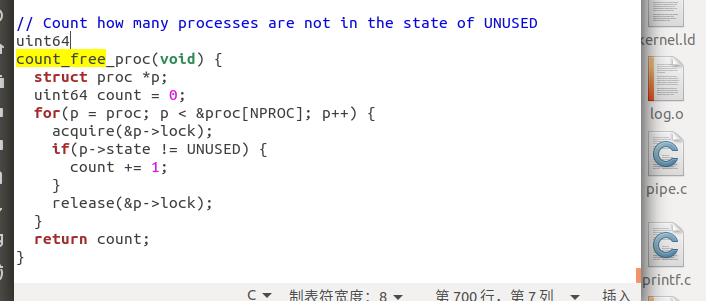
1.获得空余内存的字节数

根据实验指引, 查看kernel/kalloc.c内核文件。 可以观察到每个物理内存页的单位是PGSIZE=4096字节, 以一个单链表的形式管理空闲内存页. 整个xv6的内存地址空间是由void kinit()这个函数来定义的.。kmem->freelist指向第一个空余内存页, 然后去计算整个系统的空余内存只需要遍历一次这个单链表即可。



2.获得分配出去的进程数量

根据实验指引, 查看kernel/proc.c内核文件。 在这个文件的一开头, 我们看到xv6静态分配了最多64个进程的信息存储空间在struct proc proc[NPROC]. 同时在void procinit(void)里每一个存储空间都被初始化了, 不会有空悬指针访问的危险。所以我们遍历这个存储空间列表即可知道分配出去了多少进程。

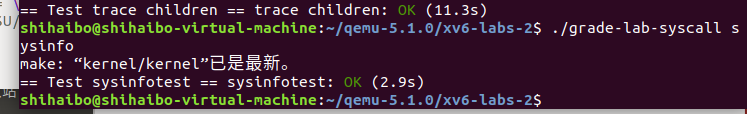


3. 将数据拷贝到用户态提供的buffer里

xv6的用户态和内核态的数据并不能直接交互，需要使用copyout函数来将内核态的数据拷贝到用户态地址上。 然后写出sysinfo系统调用内核函数。



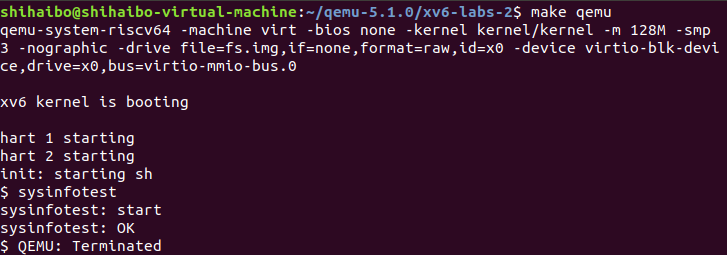
4. 把文件目标加入Makefile后进行lab批分, 顺利通过.



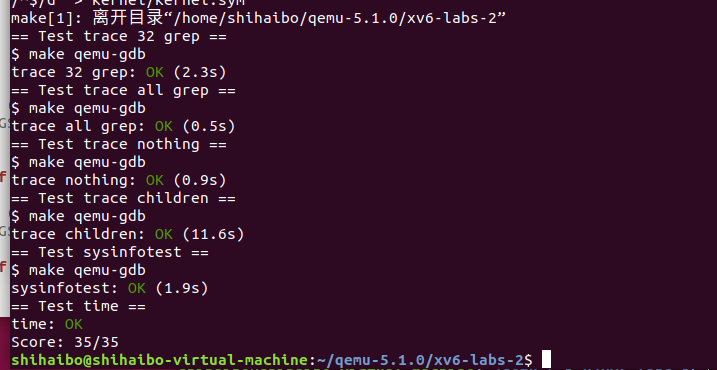
5.测试结果

指令：1. **make qemu**

2. **sysinfotest**



#### 2.3.3综合测试



### 2.4 实验中的问题和解决方法

本次实验实际上是对系统调用的深入理解，在做这次实验的过程中我遇到的问题相对比较多。在第一个 trace 的过程中，需要阅读比较多的内核 kernel 源代码，包括 syscall()等函数，只有熟悉了这些函数才能够比较好的完成实验内容。

在完成第二部分 sysinfo 的时候，由于对进程的定义仅仅停留在课本上，因此刚开始的时候难以下手。但是通过查阅相关资料，我了解到 xv6 中对进程的定义， 以及存储所有进程的变量和进程状态变量，仿照着已经存在的内核函数，终于完成了该部分。

总体来说，这次实验让我在实验一的基础上更加了解系统调用，同时也为未来的实验打下了比较扎实的基础。此外，也让我更加熟悉了写系统调用的方法， 对内核和用户也有了比较好的区分。

### 2.5 实验心得

1.系统调用的概念与作用：实验中我深入学习了系统调用的概念和作用。系统调用提供了用户程序与操作系统之间的接口，允许用户程序访问操作系统提供的服务和资源。通过实现系统调用，我更深入地理解了操作系统内核与用户空间程序之间的交互方式。

2.操作系统内核代码的结构：实验中我需要阅读和修改xv6内核的代码，添加和实现新的系统调用。通过仔细研究xv6的代码结构和各个模块之间的依赖关系，我对操作系统内核的组织结构和实现方式有了更清晰的认识。

3.系统调用的实现：实验要求我实现一些基本的系统调用，如fork、exec、wait等。在实现这些系统调用的过程中，我需要深入理解xv6内核的进程管理机制、文件系统和内存管理等相关知识，并在内核中正确地添加和修改代码。

4.用户程序与系统调用的交互：实验中我需要编写一些测试程序，通过系统调用与操作系统进行交互。这包括调用系统调用并处理返回值，以及涉及文件读写、进程管理和权限控制等功能的操作。通过编写测试程序，我更好地理解了用户程序如何使用系统调用来访问操作系统的功能。

5.调试和错误处理：在实验过程中，我遇到了一些错误和bug。通过使用GDB调试工具和仔细检查错误提示信息，我学会了如何定位和解决问题，提高了我的调试能力。

## 三 Lab: page tables

### 实验目的

1.了解 RISC-V 中的页表机制。

2.深入理解逻辑地址到物理地址的映射关系，掌握如何实现一个页表映射。

3.为每个进程实现一个页表，以更加深刻的认识到页表的重要性。

4.加深对操作系统内存管理的理解。

### 实验内容

1.定义函数 vmprint()，接受一个 pagetable\_t 参数，功能是打印页表结构。

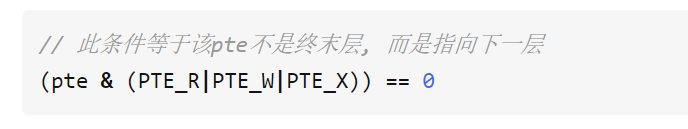
2.修改内核，使得每个进程在内核中执行的时候都使用自己的内核页表副本。

3.内核中的 copyin()读取用户指针指向的内存，它会将其翻译成物理地址。给每个进程的内核页表都添加用户映射，使得 copyin 可以直接使用用户指针。

### 实验过程

#### 3.31 Print a page table

1. 我们可以基本模仿**void freewalk(pagetable\_t pagetable)**这个函数来完成要求.



代码

*// Recursive helper*

**void** **vmprint\_helper**(pagetable\_t pagetable, **int** depth) {

**static** **char\*** indent[] **=** {

"",

"..",

".. ..",

".. .. .."

};

**if** (depth **<=** 0 **||** depth **>=** 4) {

panic("vmprint\_helper: depth not in {1, 2, 3}");

}

*// there are 2^9 = 512 PTES in a page table.*

**for** (**int** i **=** 0; i **<** 512; i**++**) {

pte\_t pte **=** pagetable[i];

**if** (pte **&** PTE\_V) { *//是一个有效的PTE*

printf("%s%d: pte %p pa %p\n", indent[depth], i, pte, PTE2PA(pte));

**if** ((pte **&** (PTE\_R**|**PTE\_W**|**PTE\_X)) **==** 0) {

*// points to a lower-level page table 并且是间接层PTE*

uint64 child **=** PTE2PA(pte);

vmprint\_helper((pagetable\_t)child, depth**+**1); *// 递归, 深度+1*

}

}

}

}

*// Utility func to print the valid*

*// PTEs within a page table recursively*

**void** **vmprint**(pagetable\_t pagetable) {

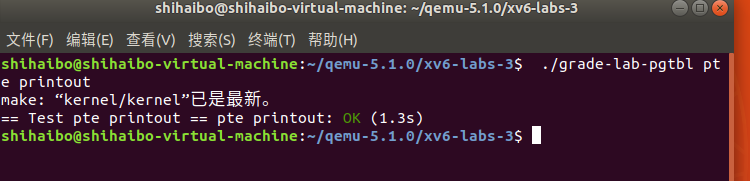
printf("page table %p\n", pagetable);

vmprint\_helper(pagetable, 1);

}

2. 编译后进行lab批分

./grade-lab-pgtbl pte printout



#### 3.3.2 a kernel page table per process

1. 为**struct proc**加入一个新的field

**# kernel/proc.h**

*// Per-process state*

**struct** proc {

uint64 tracemask; *// the sys calls this proc is tracing*

pagetable\_t kpagetable; *// the kernel table per process 专属内核页*

};

2. 模仿**kvminit**函数, 在**allocproc**函数里为新进程分配它的专属内核页

**# kernel/vm.c**

*// add a mapping to the per-process kernel page table.*

**void**

**ukvmmap**(pagetable\_t kpagetable, uint64 va, uint64 pa, uint64 sz, **int** perm)

{

**if**(mappages(kpagetable, va, sz, pa, perm) **!=** 0)

panic("ukvmmap");

}

*/\**

*\* create a direct-map page table for the per-process kernel page table.*

*\* return nullptr when kalloc fails*

*\*/*

pagetable\_t

**ukvminit**()

{

pagetable\_t kpagetable **=** (pagetable\_t) kalloc();

**if** (kpagetable **==** 0) {

**return** kpagetable;

}

memset(kpagetable, 0, PGSIZE);

*// 把固定的常数映射照旧搬运过来*

*// uart registers*

ukvmmap(kpagetable, UART0, UART0, PGSIZE, PTE\_R **|** PTE\_W);

*// virtio mmio disk interface*

ukvmmap(kpagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE\_R **|** PTE\_W);

*// CLINT*

ukvmmap(kpagetable, CLINT, CLINT, 0x10000, PTE\_R **|** PTE\_W);

*// PLIC*

ukvmmap(kpagetable, PLIC, PLIC, 0x400000, PTE\_R **|** PTE\_W);

*// map kernel text executable and read-only.*

ukvmmap(kpagetable, KERNBASE, KERNBASE, (uint64)etext**-**KERNBASE, PTE\_R **|** PTE\_X);

*// map kernel data and the physical RAM we'll make use of.*

ukvmmap(kpagetable, (uint64)etext, (uint64)etext, PHYSTOP**-**(uint64)etext, PTE\_R **|** PTE\_W);

*// map the trampoline for trap entry/exit to*

*// the highest virtual address in the kernel.*

ukvmmap(kpagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE\_R **|** PTE\_X);

**return** kpagetable;

}

**static** **struct** proc**\***

allocproc(**void**)

{

...

found:

p**->**pid **=** allocpid();

...

*// An empty user page table.*

p**->**pagetable **=** proc\_pagetable(p);

**if**(p**->**pagetable **==** 0){

freeproc(p);

release(**&**p**->**lock);

**return** 0;

}

*// Allocate the per-process kernel page table*

*// 为这个进程分配并初始化一个新的专属内核页*

p**->**kpagetable **=** ukvminit();

**if**(p**->**kpagetable **==** 0) {

freeproc(p);

release(**&**p**->**lock);

**return** 0;

}

...

}

3. 在这个新的专属内核页里, 把这个新进程的**kernel stack**映射上. 注意在**procinit**函数里, 所有进程的kernel stack的物理页都已经被分配了. 并不需要重新去分配物理页, 只需要建立映射mapping即可.

**static** **struct** proc**\***

**allocproc**(**void**)

{

...

found:

p**->**pid **=** allocpid();

...

*// An empty user page table.*

p**->**pagetable **=** proc\_pagetable(p);

**if**(p**->**pagetable **==** 0){

freeproc(p);

release(**&**p**->**lock);

**return** 0;

}

*// Allocate the per-process kernel page table*

*// 为这个进程分配并初始化一个新的专属内核页*

p**->**kpagetable **=** ukvminit();

**if**(p**->**kpagetable **==** 0) {

freeproc(p);

release(**&**p**->**lock);

**return** 0;

}

*// remap the kernel stack page per process*

*// physical address is already allocated in procinit()*

*// 每个kernel stack的虚拟地址va是已经提前决定好的 直接算出它对应的pa*

*// 然后把这个映射加入到新的专属内核页里*

uint64 va **=** KSTACK((**int**) (p **-** proc));

pte\_t pa **=** kvmpa(va);

memset((**void** **\***)pa, 0, PGSIZE); *// 刷新清空kernel stack*

ukvmmap(p**->**kpagetable, va, (uint64)pa, PGSIZE, PTE\_R **|** PTE\_W);

p**->**kstack **=** va;

...

}

4. 在**scheduler**切换进程的时候, 刷新TLB和使用的虚拟-物理页表影射base. 注意在进程切换跑完返回后, 要重新切换回全局的kernel page. 这个点让我debug了好几个小时.

**# kernel/proc.c**

**void**

**scheduler**(**void**)

{

**struct** proc **\***p;

**struct** cpu **\***c **=** mycpu();

c**->**proc **=** 0;

**for**(;;){

*// Avoid deadlock by ensuring that devices can interrupt.*

intr\_on();

**int** found **=** 0;

**for**(p **=** proc; p **<** **&**proc[NPROC]; p**++**) {

acquire(**&**p**->**lock);

**if**(p**->**state **==** RUNNABLE) {

*// Switch to chosen process. It is the process's job*

*// to release its lock and then reacquire it*

*// before jumping back to us.*

p**->**state **=** RUNNING;

c**->**proc **=** p;

*// 切换到要马上运行的新进程的内核页表*

w\_satp(MAKE\_SATP(p**->**kpagetable));

sfence\_vma();

swtch(**&**c**->**context, **&**p**->**context);

*// Process is done running for now.*

*// It should have changed its p->state before coming back.*

*// 切换回全局内核页表*

kvminithart();

c**->**proc **=** 0;

found **=** 1;

}

release(**&**p**->**lock);

}

...

}

5. 在销毁一个进程时, 回收它的内核页表. 这里需要注意的是, 我们并不需要去回收内核页表所映射到的物理地址. 因为那些物理地址, 例如device mapping, 是全局共享的. 进程专属内核表只是全局内核表的一个复制. 但是间接映射所消耗分配的物理内存是需要回收的. 举个例子, 在kernel pagetable可能有这样一个三级映射:

0x 810 (第一级) -> 0x 910 (第二级) -> 0x 1100(第三级) -> 0x 10000000L **UART0**

需要把**0x 810**, **0x 910**, **0x 1100** 回收的, 但是**UARTO**不需要回收因为是共享的.

**# kernel/vm.c**

*// Unmap the leaf node mapping*

*// of the per-process kernel page table*

*// so that we could call freewalk on that*

**void**

**ukvmunmap**(pagetable\_t pagetable, uint64 va, uint64 npages)

{

uint64 a;

pte\_t **\***pte;

**if**((va **%** PGSIZE) **!=** 0)

panic("ukvmunmap: not aligned");

**for**(a **=** va; a **<** va **+** npages**\***PGSIZE; a **+=** PGSIZE){

**if**((pte **=** walk(pagetable, a, 0)) **==** 0)

**goto** clean;

**if**((**\***pte **&** PTE\_V) **==** 0)

**goto** clean;

**if**(PTE\_FLAGS(**\***pte) **==** PTE\_V)

panic("ukvmunmap: not a leaf");

clean:

**\***pte **=** 0;

}

}

*// Recursively free page-table pages similar to freewalk*

*// not need to already free leaf node*

*// 和freewalk一模一样, 除了不再出panic错当一个page的leaf还没被清除掉*

*// 因为当我们free pagetable和kpagetable的时候*

*// 只有1份物理地址, 且原本free pagetable的函数会负责清空它们*

*// 所以这个函数只需要把在kpagetable里所有间接mapping清除即可*

**void**

**ufreewalk**(pagetable\_t pagetable)

{

*// there are 2^9 = 512 PTEs in a page table.*

**for**(**int** i **=** 0; i **<** 512; i**++**){

pte\_t pte **=** pagetable[i];

**if**((pte **&** PTE\_V) **&&** (pte **&** (PTE\_R**|**PTE\_W**|**PTE\_X)) **==** 0){

*// this PTE points to a lower-level page table.*

uint64 child **=** PTE2PA(pte);

ufreewalk((pagetable\_t)child);

pagetable[i] **=** 0;

}

pagetable[i] **=** 0;

}

kfree((**void\***)pagetable);

}

*// helper function to first free all leaf mapping*

*// of a per-process kernel table but do not free the physical address*

*// and then remove all 3-levels indirection and the physical address*

*// for this kernel page itself*

**void** **freeprockvm**(**struct** proc**\*** p) {

pagetable\_t kpagetable **=** p**->**kpagetable;

*// reverse order of allocation*

*// 按分配顺序的逆序来销毁映射, 但不回收物理地址*

ukvmunmap(kpagetable, p**->**kstack, PGSIZE**/**PGSIZE);

ukvmunmap(kpagetable, TRAMPOLINE, PGSIZE**/**PGSIZE);

ukvmunmap(kpagetable, (uint64)etext, (PHYSTOP**-**(uint64)etext)**/**PGSIZE);

ukvmunmap(kpagetable, KERNBASE, ((uint64)etext**-**KERNBASE)**/**PGSIZE);

ukvmunmap(kpagetable, PLIC, 0x400000**/**PGSIZE);

ukvmunmap(kpagetable, CLINT, 0x10000**/**PGSIZE);

ukvmunmap(kpagetable, VIRTIO0, PGSIZE**/**PGSIZE);

ukvmunmap(kpagetable, UART0, PGSIZE**/**PGSIZE);

ufreewalk(kpagetable);

}

**======================================================================**

**# kernel/proc.c**

*// free a proc structure and the data hanging from it,*

*// including user pages.*

*// p->lock must be held.*

**static** **void**

freeproc(**struct** proc **\***p)

{

**if**(p**->**trapframe)

kfree((**void\***)p**->**trapframe);

p**->**trapframe **=** 0;

**if**(p**->**pagetable)

proc\_freepagetable(p**->**pagetable, p**->**sz);

p**->**pagetable **=** 0;

p**->**sz **=** 0;

p**->**pid **=** 0;

p**->**parent **=** 0;

p**->**name[0] **=** 0;

p**->**chan **=** 0;

p**->**killed **=** 0;

p**->**xstate **=** 0;

p**->**state **=** UNUSED;

**if** (p**->**kpagetable) {

freeprockvm(p);

p**->**kpagetable **=** 0;

}

**if** (p**->**kstack) {

p**->**kstack **=** 0;

}

}

#### 3.3.3 simplify copyin/copyinstr

1.首先写一个helper函数, 来将一段内存映射从**pagetable**复制到**kpagetable**.

**# kernel/vm.c**

*// Same as mappages without panic on remapping*

*// 和mappages一模一样, 只不过不再panic remapping, 直接强制复写*

**int** **umappages**(pagetable\_t pagetable, uint64 va, uint64 size, uint64 pa, **int** perm) {

uint64 a, last;

pte\_t **\***pte;

a **=** PGROUNDDOWN(va);

last **=** PGROUNDDOWN(va **+** size **-** 1);

**for**(;;){

**if**((pte **=** walk(pagetable, a, 1)) **==** 0)

**return** **-**1;

**\***pte **=** PA2PTE(pa) **|** perm **|** PTE\_V;

**if**(a **==** last)

**break**;

a **+=** PGSIZE;

pa **+=** PGSIZE;

}

**return** 0;

}

*// copying from old page to new page from*

*// begin in old page to new in old page*

*// and mask off PTE\_U bit*

*// 将从begin到end的虚拟地址的映射, 从oldpage复制到newpage*

**int**

**pagecopy**(pagetable\_t oldpage, pagetable\_t newpage, uint64 begin, uint64 end) {

pte\_t **\***pte;

uint64 pa, i;

uint flags;

begin **=** PGROUNDUP(begin);

**for** (i **=** begin; i **<** end; i **+=** PGSIZE) {

**if** ((pte **=** walk(oldpage, i, 0)) **==** 0)

panic("pagecopy walk oldpage nullptr");

**if** ((**\***pte **&** PTE\_V) **==** 0)

panic("pagecopy oldpage pte not valid");

pa **=** PTE2PA(**\***pte);

flags **=** PTE\_FLAGS(**\***pte) **&** (**~**PTE\_U); *// 把U flag抹去*

**if** (umappages(newpage, i, PGSIZE, pa, flags) **!=** 0) {

**goto** err;

}

}

**return** 0;

err:

uvmunmap(newpage, 0, i **/** PGSIZE, 1);

**return** **-**1;

}

2.在**fork()**, **exec()**, **sbrk()** 和**userinit()**的相应位置进行**pagetable**和**kpagetale**的同步.

**fork()**

**# kernel/proc.c**

*// Create a new process, copying the parent.*

*// Sets up child kernel stack to return as if from fork() system call.*

**int**

**fork**(**void**)

{

...

*// Copy user memory from parent to child.*

**if**(uvmcopy(p**->**pagetable, np**->**pagetable, p**->**sz) **<** 0){

freeproc(np);

release(**&**np**->**lock);

**return** **-**1;

}

np**->**sz **=** p**->**sz;

**if** (pagecopy(np**->**pagetable, np**->**kpagetable, 0, np**->**sz) **!=** 0) {

freeproc(np);

release(**&**np**->**lock);

**return** **-**1;

}

np**->**parent **=** p;

...

**return** pid;

}

**exec()**

**# kernel/exec.c**

**int**

**exec**(**char** **\***path, **char** **\*\***argv)

{

...

*// Commit to the user image.*

oldpagetable **=** p**->**pagetable;

p**->**pagetable **=** pagetable;

p**->**sz **=** sz;

p**->**trapframe**->**epc **=** elf.entry; *// initial program counter = main*

p**->**trapframe**->**sp **=** sp; *// initial stack pointer*

proc\_freepagetable(oldpagetable, oldsz);

*// 复制新的kernel page并刷新TLB*

**if** (pagecopy(p**->**pagetable, p**->**kpagetable, 0, p**->**sz) **!=** 0) {

**goto** bad;

}

*// 因为load进来了新的program, 刷新一下内存映射*

w\_satp(MAKE\_SATP(p**->**kpagetable));

sfence\_vma();

...

**return** argc; *// this ends up in a0, the first argument to main(argc, argv)*

}

**sbrk()**

**# kernel/proc.c**

*// Grow or shrink user memory by n bytes.*

*// Return 0 on success, -1 on failure.*

**int**

**growproc**(**int** n)

{

uint sz;

**struct** proc **\***p **=** myproc();

sz **=** p**->**sz;

**if**(n **>** 0){

*// 内核页的虚拟地址不能溢出PLIC*

**if** (sz **+** n **>** PLIC **||** (sz **=** uvmalloc(p**->**pagetable, sz, sz **+** n)) **==** 0) {

**return** **-**1;

}

**if** (pagecopy(p**->**pagetable, p**->**kpagetable, p**->**sz, sz) **!=** 0) {

*// 增量同步[old size, new size]*

**return** **-**1;

}

} **else** **if**(n **<** 0){

sz **=** uvmdealloc(p**->**pagetable, sz, sz **+** n);

**if** (sz **!=** p**->**sz) {

*// 缩量同步[new size, old size]*

uvmunmap(p**->**kpagetable, PGROUNDUP(sz), (PGROUNDUP(p**->**sz) **-** PGROUNDUP(sz)) **/** PGSIZE, 0);

}

}

ukvminithard(p**->**kpagetable);

p**->**sz **=** sz;

**return** 0;

}

**userinit()**

**# kernel/proc.c**

*// Set up first user process.*

**void**

**userinit**(**void**)

{

...

uvminit(p**->**pagetable, initcode, **sizeof**(initcode));

p**->**sz **=** PGSIZE;

pagecopy(p**->**pagetable, p**->**kpagetable, 0, p**->**sz);

*// prepare for the very first "return" from kernel to user.*

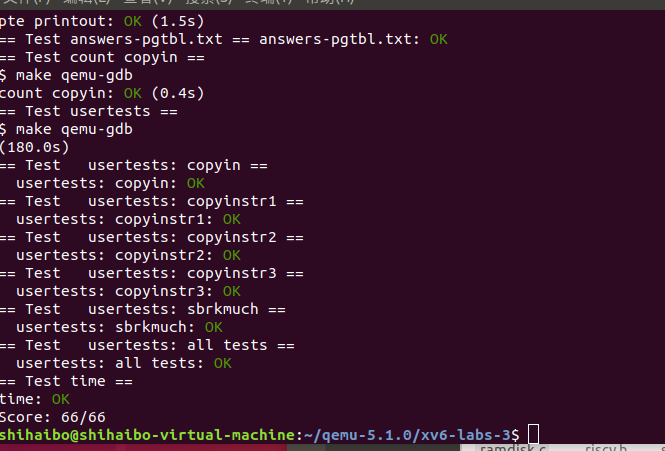
p**->**trapframe**->**epc **=** 0; *// user program counter*

p**->**trapframe**->**sp **=** PGSIZE; *// user stack pointer*

...

}

#### 3.3.4 综合测试



### 3.4 实验中的问题和解决方法

这个实验可以说是所有实验中花费时间最多的一个实验了。因为这次实验涉及到了操作系统中一个重要的概念，就是页表结构。同时，还涉及到了逻辑地址到物理地址之间的转换，我在复习了相关操作系统知识才得以顺利解决问题。

在本次实验的过程中，还涉及到了操作系统中地址的定位，在完成的过程中还要注意越界判断等。所以可以说，这次实验的完成是对能力的一个很好的提升。

### 3.5 实验心得

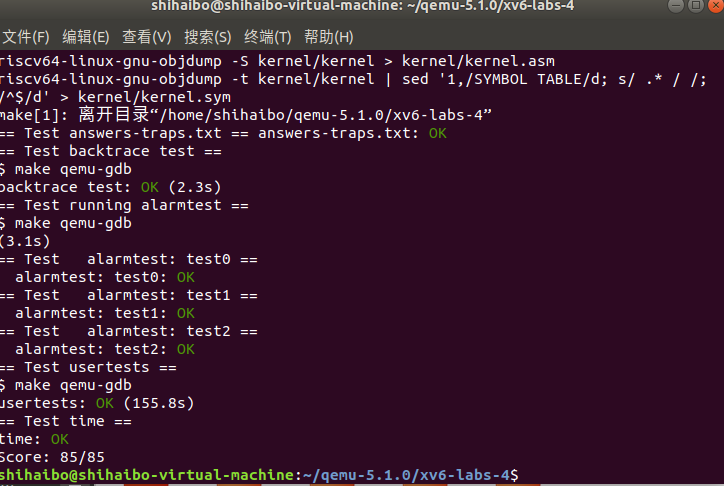
1. 虚拟内存和页表的概念：实验中我学习了虚拟内存和页表的概念，并了解了它们在操作系统中的作用。虚拟内存提供了一种抽象，使得每个进程都能拥有自己独立的地址空间，而页表则负责将虚拟地址映射到物理地址，实现地址转换。

2. 理解xv6操作系统中的页表实现：实验要求我深入研究xv6内核中的页表实现方式，并实现一些与页表相关的功能。通过阅读和修改xv6的代码，我加深了对页表数据结构和操作的理解，以及页表在内存管理中的作用。

3. 内存管理和页面替换算法：实验中我需要实现页面替换算法，以处理当物理内存不足时的页面置换问题。这要求我深入了解页面置换算法的原理和性能特征，并在实际代码中正确地实现这些算法。通过实现页面替换算法，我深入理解了内存管理的复杂性和性能优化的重要性。

4. 调试和错误处理：在实验过程中，我遇到了一些错误和bug。通过使用GDB进行代码调试，并运行一些测试程序来验证我的修改，我学会了如何快速定位和解决问题，提高了我的调试能力。

5. 性能分析和优化：实验中我需要使用一些性能分析工具来评估我的页表实现和页面替换算法的性能。通过分析性能数据和优化代码，我学会了如何提高系统的效率和响应性。



## 四 Lab: Traps

### 4.1 实验目的

1. 进一步了解系统调用所发挥的重要作用。
2. 掌握通过中断进行系统调用的过程及其所发挥的作用。
3. 了解简单的汇编语言，知道汇编语言是如何发挥作用的。
4. 理解 xv6 中的堆栈，同时试着实现一个用户级中断处理。

### 4.2 实验内容

1. 了解 RISC-V 程序集，阅读 call.asm 中的函数 g、f 和 main 的代码，并回答问题。
2. 完成 Backtrace 函数，当发生错误的时候查看当前堆栈中的系统调用。
3. 添加系统调用 sigalarm，当用户程序运行了 n 个 ticks 后，触发回调函数。

### 4.3 实验过程

#### 4.3.1 Rise-v assembly

按实验说明,**make fs.img** 编译出**user/call.asm**来进行分析.

1. Which registers contain arguments to functions**?**

For example, which **register** holds 13 in main's call to printf**?**

根据risc**-**v的手册, a0**-**a7都是可以存放函数调用的参数的.

在main函数的汇编代码中:

000000000000001c **<**main**>:**

**void** main(**void**) {

1c: 1141 addi sp,sp,**-**16

1e: e406 sd ra,8(sp)

20**:** e022 sd s0,0(sp)

22**:** 0800 addi s0,sp,16

printf("%d %d\n", f(8)**+**1, 13);

24**:** 4635 li a2,13

26**:** 45b1 li a1,12

28**:** 00000517 auipc a0,0x0

2c: 7b050513 addi a0,a0,1968 # 7d8 **<**malloc**+**0xe8**>**

30**:** 00000097 auipc ra,0x0

34**:** 608080e7 jalr 1544(ra) # 638 **<**printf**>**

exit(0);

38**:** 4501 li a0,0

3a: 00000097 auipc ra,0x0

3e: 274080e7 jalr 628(ra) # 2ae **<**exit**>**

可以看到, 寄存器a2存放了13这个值

**-------------------------------------------------------------------**

2. Where is the call to function f in the assembly code **for** main**?**

Where is the call to g**?** (Hint: the compiler may **inline** functions.)

一整个函数调用链f(g(8))**+**1都被编译器直接优化成inline了并算出了答案**=**12, 放置在了a1寄存器里

由这一行汇编源码可以看出**:**

26**:** 45b1 li a1,12

**-------------------------------------------------------------------**

3. At what address is the function printf located**?**

由这一行汇编源码**:**

0000000000000638 **<**printf**>**

可以得知printf在0x638的位置

但是如何跳转到0x638的位置的呢**?** 来看一下这两行

30**:** 00000097 auipc ra,0x0

34**:** 608080e7 jalr 1544(ra) # 638 **<**printf**>**

auipcj将0右移12位后赋值给ra并加上当前的指令位置, 所有ra值为0x30.

下一行1544(ra) **=** 1544 **+** 0x30 **=** 0x638 跳转到printf了就

**-------------------------------------------------------------------**

4. What value is in the **register** ra just after the jalr to printf in main**?**

jalr这个指令会把下一个要执行的指令的地址压入ra, 及0x34**+**4 **=** 0x38

**-------------------------------------------------------------------**

5. Run the following code.

**unsigned** **int** i **=** 0x00646c72;

printf("H%x Wo%s", 57616, **&**i);

What is the output**?** Here's an ASCII table that maps bytes to characters.

The output depends on that fact that the RISC**-**V is little**-**endian.

If the RISC**-**V were instead big**-**endian what would you set i to in order to yield the same output**?**

Would you need to change 57616 to a different value**?**

输出的答案是 HE110 World

57616的hex是0x0000E110

i**=**0x00646c72, 字节翻译是0dlr. 因为RISC**-**V是little**-**endian, 存放在连续内存里的顺序即是rld0.

如果RISC**-**V是big**-**endian, 57616不用变, i需要变成0x726c6400

**-------------------------------------------------------------------**

6. In the following code, what is going to be printed after 'y**=**'**?**

(note: the answer is not a specific value.) Why does this happen**?**

printf("x=%d y=%d", 3);

printf的format字符串在寄存器a0, 3在寄存器a1, 所以当试图print y的时候, 留在寄存器a2的某个随机值就会被print出来

#### 4.3.2 Backtrace

1. 首先，我应该在 defs.h 中声明 backtrace()：

void **backtrace**(void);

1. 通过阅读题目，我获得了一个定义在 ricsv.h 中用来获取当前栈帧指针的函数：

static inline **uint64 r\_fp**()

{

**uint64** x;

asm volatile("mv %0, s0" : "=r" (x) ); return x;

}

1. 需要注意到，下一个栈帧指针地址为 fp-8，返回值地址为 fp-16。基于此，利用循环就可以完成：

void

**backtrace**(void){ **printf**("backtrace:\n");

*//get the start location*

**uint64** curLoc=**r\_fp**();

**uint64** upLine=**PGROUNDUP**(curLoc);

while(curLoc<upLine){ **printf**("%p\n",\*((**uint64**\*)(curLoc-8))); curLoc=\*((**uint64**\*)(curLoc-16));

}

}

#### 4.3.3 alarm

1.首先在每个struct proc里加入新的三个需要管理的fields.

**struct** proc {

**struct** spinlock lock;

uint64 tracemask;

pagetable\_t kpagetable;

**int** alarm\_period;

**void** (**\***alarm\_handler)();

**int** ticks\_since\_last\_alarm;

};

在用户呼叫sigalarm函数时, 把提供的参数记录进struct proc里.

uint64

**sys\_sigalarm**(**void**) {

**struct** proc **\***my\_proc **=** myproc();

**int** period;

**if** (argint(0, **&**period) **<** 0)

**return** **-**1;

uint64 p;

**if**(argaddr(1, **&**p) **<** 0)

**return** **-**1;

my\_proc**->**alarm\_period **=** period;

my\_proc**->**alarm\_handler **=** (**void** (**\***)()) p;

my\_proc**->**ticks\_since\_last\_alarm **=** 0;

**return** 0;

}

并且在allocproc函数里进行初始化这3个fields.

**# kernel/proc.c**

*// Look in the process table for an UNUSED proc.*

*// If found, initialize state required to run in the kernel,*

*// and return with p->lock held.*

*// If there are no free procs, or a memory allocation fails, return 0.*

**static** **struct** proc**\***

**allocproc**(**void**)

{

**struct** proc **\***p;

*// Zero initializes the tracemask for a new process*

p**->**tracemask **=** 0;

*// Zero initializes the alarm releated fields*

p**->**alarm\_period **=** 0;

p**->**alarm\_handler **=** 0;

p**->**ticks\_since\_last\_alarm **=** 0;

**return** p;

}

最后, 我们需要修改一下kernel/trap.c, 在一个alarm超时, 需要执行用户所提供的函数时, 注意我们并不是在trap函数里直接执行, 因为内核和用户的内存地址映射是不一样的. 我们修改进入trap后所记录下来的trapframe的program counter, 使得从trap中返回到用户态后, 优先跳转回需要执行的alarm函数.

**# kernel/trap.c**

*//*

*// handle an interrupt, exception, or system call from user space.*

*// called from trampoline.S*

*//*

**void**

**usertrap**(**void**)

{

**int** which\_dev **=** 0;

*// give up the CPU if this is a timer interrupt.*

**if**(which\_dev **==** 2) {

p**->**ticks\_since\_last\_alarm **+=** 1;

**if** (p**->**alarm\_period **!=** 0 **&&** p**->**ticks\_since\_last\_alarm **==** p**->**alarm\_period) {

*// 设立返回到用户态后到跳转指令地址*

*// jump to the alarm handler when returning back to user space*

p**->**trapframe**->**epc **=** (uint64)p**->**alarm\_handler; *// 修改跳转位置到alarm函数*

p**->**ticks\_since\_last\_alarm **=** 0;

}

yield();

}

usertrapret();

}

至此, alarm的第一部分完成. 我们运行alarmtest, 看到成功通过的test0发出了一个alarm.

init: starting sh

$ ./alarmtest

test0 start

...................alarm!

test0 passed

test0 passed

usertrap**()**: unexpected scause 0x000000000000000d pid**=**3

sepc**=**0x0000000000000166 stval**=**0x0000000000003018

$

2. 在跳转到alarm函数前, 我们需要完整的复制整个**trapframe**, 在sigreturn函数里再重新恢复这些所有的寄存器.

首先我们为**struct proc**增加新的两个field, 并为之初始化和析构.

**# kernel/proc.h**

*// Per-process state*

**struct** proc {

**struct** spinlock lock;

... 省略...

**struct** trapframe **\***alarmframe;*// data page to restore all register when going back from alarm handler*

**int** alarm\_period; *// the alarm period set*

**void** (**\***alarm\_handler)(); *// the alarm function handler*

**int** ticks\_since\_last\_alarm; *// how many ticks has elapsed since last alarm*

**int** inalarm; *// if the alarm handler is going on*

};

**---------------------------------------------------------------------**

**# kernel/proc.c**

*// Look in the process table for an UNUSED proc.*

*// If found, initialize state required to run in the kernel,*

*// and return with p->lock held.*

*// If there are no free procs, or a memory allocation fails, return 0.*

**static** **struct** proc**\***

allocproc(**void**)

{

p**->**pid **=** allocpid();

*// Allocate a trapframe page.*

**if**((p**->**trapframe **=** (**struct** trapframe **\***)kalloc()) **==** 0){

release(**&**p**->**lock);

**return** 0;

}

*// Allocate a alarmframe page. 为这个备份alarmframe分配1页物理空间*

**if**((p**->**alarmframe **=** (**struct** trapframe **\***)kalloc()) **==** 0){

release(**&**p**->**lock);

**return** 0;

}

*// Zero initializes the alarm releated fields*

p**->**alarm\_period **=** 0;

p**->**alarm\_handler **=** 0;

p**->**ticks\_since\_last\_alarm **=** 0;

p**->**inalarm **=** 0;

**return** p;

}

*// free a proc structure and the data hanging from it,*

*// including user pages.*

*// p->lock must be held.*

**static** **void**

freeproc(**struct** proc **\***p)

{

**if**(p**->**trapframe)

kfree((**void\***)p**->**trapframe);

**if** (p**->**alarmframe) *// 释放物理内存*

kfree((**void** **\***)p**->**alarmframe);

p**->**alarmframe **=** 0;

}

紧接着, 我们在**kernel/trap.c**的函数里, 每当要触发用户设置的alarm函数时, 保存当前所有的寄存器到**p->alarmframe**里, 并设立一个**p->inalarm**旗帜使得在上一个alarm函数还没处理完前, 不会重复进入这个alarm函数.

**# kernel/trap.c**

*//*

*// handle an interrupt, exception, or system call from user space.*

*// called from trampoline.S*

*//*

**void**

**usertrap**(**void**)

{

**int** which\_dev **=** devintr();

*// give up the CPU if this is a timer interrupt.*

**if**(which\_dev **==** 2) {

p**->**ticks\_since\_last\_alarm **+=** 1;

**if** (p**->**inalarm **==** 0 **&&** p**->**alarm\_period **!=** 0 **&&** p**->**ticks\_since\_last\_alarm **==** p**->**alarm\_period) {

p**->**inalarm **=** 1; *// 避免重复进入*

**\***p**->**alarmframe **=** **\***p**->**trapframe; *// 备份当前寄存器*

*// save all the trapframe to alarmframe for later restore*

*// jump to the alarm handler when returning back to user space*

p**->**trapframe**->**epc **=** (uint64)p**->**alarm\_handler; *// 修改跳转位置到alarm函数*

}

yield();

}

usertrapret();

}

最后, 在**sigreturn**函数里, 我们重新取回被alarm中断前到所有寄存器状态并去掉**p->inalarm**旗帜, 使得可以再次alarm中断.

**# kernel/sysproc.c**

uint64

**sys\_sigreturn**(**void**) {

**struct** proc**\*** p **=** myproc();

**if** (p**->**inalarm) {

p**->**inalarm **=** 0;

**\***p**->**trapframe **=** **\***p**->**alarmframe;

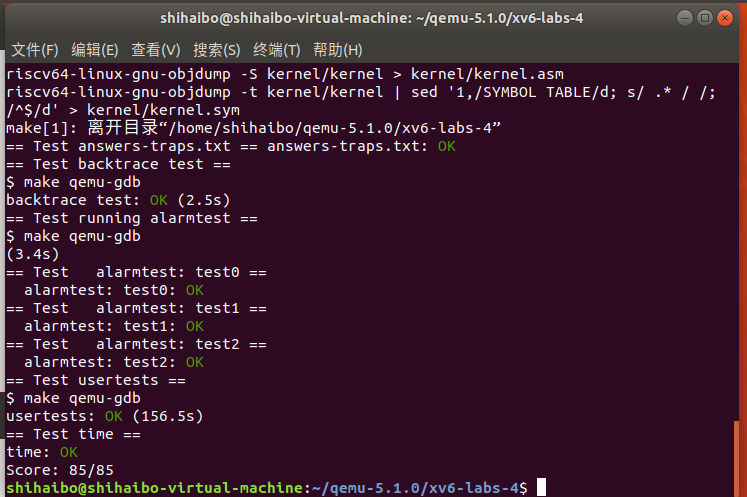
p**->**ticks\_since\_last\_alarm **=** 0;

}

**return** 0;

}

#### 4.3.4综合测试



### 4.4 实验中的问题和解决方法

1.中断处理错误：实验要求实现处理外部中断（如定时器中断、键盘中断等）的处理程序。在处理中断时，可能会出现中断响应不及时或中断处理程序错误的情况。要解决这些问题，需要正确设置中断向量表，并确保中断处理程序能够及时响应和处理中断。

2.上下文切换错误：在处理异常和中断时，需要进行上下文切换操作，保存当前进程的上下文，并切换到新的上下文执行。如果上下文切换过程出现错误，可能导致进程状态混乱或数据丢失。要解决这些问题，需要正确实现上下文切换机制，并确保保存和恢复上下文的正确性。

### 4.5 实验心得

这次实验主要是针对于操作系统中一个非常重要的中断机制而展开的。中断是指计算机运行过程中，出现某些意外情况需主机干预时，机器能自动停止正在运行的程序并转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行。

在本次实验的 alarm 中，我实际上就是通过实现中断现场的保存和处理结束后现场的恢复，实现了一个中断的过程。在这个过程中，其实我遇到的问题挺多的，包括不知道哪些状态需要保存下来、保存到什么地方去、保存之后怎么办等等。但是真正把这一个实验做下来之后，我对中断也有了更加深刻的理解，对操作系统中断机制也有了更好的认识。

## 五 Lab: Lazy page allocation

### 5.1 实验目的

1. 通过该实验了解操作系统中常用的策略惰性空间分配。
2. 体会到惰性空间分配对提高操作系统性能所发挥的巨大作用。
3. 尝试完成惰性空间分配，并体会它所带来的性能上的提升。

### 5.2 实验内容

1. 从 sbrk()系统调用实现中删除页面分配，即 sysproc.c 中的函数 sys\_sbrk()。
2. 修改 trap.c 中的代码，通过在错误地址映射新分配的物理内存页来响应来自用户空间的页面错误，然后返回到用户空间让进程继续执行。
3. lazytests 用于测试一些可能会给惰性内存分配器带来压力的特定情况，修改内核代码以通过全部测试。

### 5.3 实验过程

#### 5.3.1 Eliminate allocation from sbrk

1.我们需要修改一下**sbrk()**函数: 在用户申请内存空间时, 只是"虚假"地增加这个进程的大小而并不去分配物理页, 为了之后用户在真正使用这个地址时, 可以发生缺页中断来让内核介入进行实际的页分配。

uint64

**sys\_sbrk**(**void**)

{

**int** addr;

**int** n;

**if**(argint(0, **&**n) **<** 0)

**return** **-**1;

addr **=** myproc()**->**sz;

myproc()**->**sz **+=** n; *// 虚假地增加size而不物理分配*

*// if(growproc(n) < 0)*

*// return -1;*

**return** addr;

}

#### 5.3.2 lazy allocation

2.在**usertrap**函数里加入额外的处理逻辑来应对缺页中断异常(**13**号或**15**号). 注意与课程视频里教授给出的基本代码不同的是, 由于在**lab3**里我们为每个进程都引入了独立的内核页, 且内核页从**地址0**到**地址PLIC**是和进程的**pagetable**同步的, 我们在分配完一个物理页后, 需要进行2次**mappages**操作.

**void**

**usertrap**(**void**)

{

} **else** **if** (r\_scause() **==** 13 **||** r\_scause() **==** 15) {

printf("page fault trap: signal %d at address %p\n", r\_scause(), r\_stval());

uint64 va **=** PGROUNDDOWN(r\_stval()); *// beginning of a 4KB page*

uint64 pa **=** (uint64) kalloc();

**if** (pa **==** 0) {

p**->**killed **=** 1;

} **else** {

memset((**void** **\***)pa, 0, PGSIZE);

**if** (umappages(p**->**pagetable, va, PGSIZE, pa, PTE\_W**|**PTE\_X**|**PTE\_R**|**PTE\_U) **!=** 0) {

kfree((**void** **\***)pa);

p**->**killed **=** 1;

}

**if** (p**->**killed **==** 0) {

**if** (umappages(p**->**kpagetable, va, PGSIZE, pa, PTE\_W **|** PTE\_X **|** PTE\_R) **!=** 0) {

kfree((**void** **\***)pa);

uvmunmap(p**->**pagetable, va, 1, 0);

p**->**killed **=** 1;

}

}

}

} **else** {

printf("usertrap(): unexpected scause %p pid=%d\n", r\_scause(), p**->**pid);

printf(" sepc=%p stval=%p\n", r\_sepc(), r\_stval());

p**->**killed **=** 1;

}

...省略...

usertrapret();

**return**;

}

#### 5.3.3 lazytests and ysertests

1.首先把**lazy allocation**的函数单独抽出来, 作更好的模块化处理.

**int** **lazyvalidate**(**struct** proc**\*** p, uint64 va) {

**if** (va **>** p**->**sz **||** va **<** p**->**trapframe**->**sp) {

**return** **-**1;

}

**if** (walkaddr(p**->**pagetable, va) **!=** 0) {

**return** 0;

}

va **=** PGROUNDDOWN(va);

uint64 pa **=** (uint64) kalloc();

**if** (pa **==** 0) {

**return** **-**1;

}

memset((**void** **\***)pa, 0, PGSIZE);

**if** (umappages(p**->**pagetable, va, PGSIZE, pa, PTE\_W**|**PTE\_X**|**PTE\_R**|**PTE\_U) **!=** 0) {

kfree((**void** **\***)pa);

**return** **-**1;

}

**if** (umappages(p**->**kpagetable, va, PGSIZE, pa, PTE\_W **|** PTE\_X **|** PTE\_R) **!=** 0) {

kfree((**void** **\***)pa);

uvmunmap(p**->**pagetable, va, 1, 0);

**return** **-**1;

}

**return** 0;

}

2.修改一下usertrap使得它使用这个封装起来的lazy allocation函数:

**void**

**usertrap**(**void**)

{

**int** which\_dev **=** devintr();

...省略...

} **else** **if** (r\_scause() **==** 13 **||** r\_scause() **==** 15) {

**if** (lazyvalidate(p, r\_stval()) **!=** 0) {

p**->**killed **=** 1;

**goto** killed;

}

} **else** {

printf("usertrap(): unexpected scause %p pid=%d\n", r\_scause(), p**->**pid);

printf(" sepc=%p stval=%p\n", r\_sepc(), r\_stval());

p**->**killed **=** 1;

**goto** killed;

}

...省略...

killed:

**if** (p**->**killed)

exit(**-**1);

}

3.由于lab延续了lab的copyin\_new和copyinstr\_new那一套, 每一个进程有独立的内核页的设定, 不能像网上大部分的解析一样直接修改**walkaddr**函数. 首先在内核态陷入缺页中断异常后, 我们也需要进行处理物理页分配.

**void**

**kerneltrap**()

{

**if** (scause **==** 13 **||** scause **==** 15) {

lazyvalidate(myproc(), r\_stval());

} **else** **if**((which\_dev **=** devintr()) **==** 0) {

printf("the faulting process is pid=%d\n", myproc()**->**pid);

printf("scause %p\n", scause);

printf("sepc=%p stval=%p\n", r\_sepc(), r\_stval());

panic("kerneltrap");

}

}

4.**copyout**函数原本在**walkaddr**返回**0**后会直接**return -1**, 这会造成系统调用**write**返回**-1**. 所以我们在那里也需要加入**lazy allocation**.

**int**

**copyout**(pagetable\_t pagetable, uint64 dstva, **char** **\***src, uint64 len)

{

uint64 n, va0, pa0;

**while**(len **>** 0){

va0 **=** PGROUNDDOWN(dstva);

lazyvalidate(myproc(), va0);

pa0 **=** walkaddr(pagetable, va0);

**if**(pa0 **==** 0)

**return** **-**1;

n **=** PGSIZE **-** (dstva **-** va0);

**if**(n **>** len)

n **=** len;

memmove((**void** **\***)(pa0 **+** (dstva **-** va0)), src, n);

len **-=** n;

src **+=** n;

dstva **=** va0 **+** PGSIZE;

}

**return** 0;

}

5.在测试的时候我发现**copyin\_new**似乎有个边界值的小bug. 稍加修改后可以通过测试.

**int**

**copyin\_new**(pagetable\_t pagetable, **char** **\***dst, uint64 srcva, uint64 len)

{

**struct** proc **\***p **=** myproc();

**if** (srcva **>** p**->**sz **||** srcva**+**len **>** p**->**sz **||** srcva**+**len **<** srcva)

**return** **-**1;

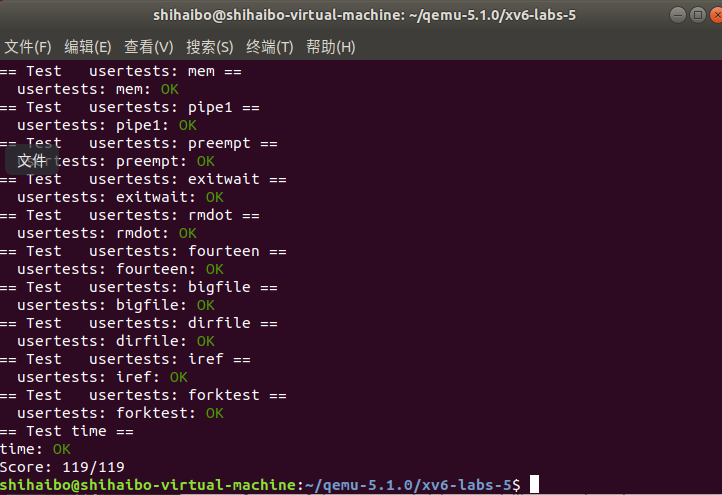
memmove((**void** **\***) dst, (**void** **\***)srcva, len);

stats.ncopyin**++**;

**return** 0;

}

#### 5.3.4综合测试



### 5.4 实验中的问题和解决方法

1.页面映射错误：实验要求实现延迟页面分配的机制，只在需要时才分配物理页面。在处理页面映射时，可能会出现错误的映射或映射不一致的情况。要解决这些问题，需要正确处理页面分配和映射的逻辑，并确保页面映射与实际需要相匹配。

2.内存访问错误：实验可能要求修改页面分配策略或修改页表项。在修改这些内容时，可能会导致内存访问错误，如无效的内存访问或内存泄漏。要解决这些问题，需要仔细检查页面分配和访问的逻辑，并确保正确地读写内存。

### 5.5 实验心得

在这个实验中，我真正实现了 UNIX 操作系统中为了提高效率而采用的一个策略，即 Lazy Allocation 惰性分配。这个实验实际上是实验三的拓展，只有掌握了实验三的页表分配机制才能够比较好的完成本次实验。在实验中由于实验3原理的理解不深刻做了太多无用功，又回去查看实验三的相关讲解资料才得以解决。

此外，本次实验中也更加深刻的体会到了用户空间和内核空间之间的区别。例如第二部分修改了 usertrap 还不够，还需要在第三部分中修改 kernel 中的异常处理。只有两部分结合起来，才是更好的操作系统。

## 六 Lab: Copy on write

### 6.1 实验目的

1. 理解什么是 Copy On Write（COW），其思想和作用是什么。
2. 熟悉简单 fork() 系统调用的工作机制，提出一种 COW 的实现手段。
3. 理解 xv6 中的 fork 函数，并修改内核程序，实现 COW。

### 6.2 实验内容

1. 修改 xv6 内核代码，使系统支持 Copy on Write 功能。虽然只有一项，但是直接完成该实验是十分困难的，因此我将实验内容拆分为了三个步骤。前两个步骤帮助我理解这个实验的内容，第三个步骤才着手修改 xv6 的源代码。
2. 完善后的实验内容：
3. 了解 Unix 中的 fork() 系统调用的功能以及实现机制，重点了解调用 fork()
4. 过程中的内存分配流程。
5. 阅读 xv6 源码，理解 fork 函数以及父子进程拷贝内容的机制。
6. 根据提示中给出的方案啊，修改 xv6 内核使系统支持 COW。

### 6.3 实验过程

#### 6.3.1 implement copy-on-write

因为一个物理页有可能同时有多个进程的虚拟页指向它, 在**freepage**的时候需要考虑到这一点. 我们需要一个计数器, 只有当一个物理页的计数器降为**0**后才真正的释放它.

先以一个静态列表的形式存储计数器, 并对**kalloc**和**kfree**进行小修改.

**int** cowcount[PHYSTOP**/**PGSIZE];

**void**

**freerange**(**void** **\***pa\_start, **void** **\***pa\_end)

{

**char** **\***p;

p **=** (**char\***)PGROUNDUP((uint64)pa\_start);

**for**(; p **+** PGSIZE **<=** (**char\***)pa\_end; p **+=** PGSIZE) {

cowcount[PA2INDEX(p)] **=** 1;

kfree(p);

}

}

**void**

**kfree**(**void** **\***pa)

{

**struct** run **\***r;

**if**(((uint64)pa **%** PGSIZE) **!=** 0 **||** (**char\***)pa **<** end **||** (uint64)pa **>=** PHYSTOP)

panic("kfree");

*// 需要加锁保证原子性*

acquire(**&**kmem.lock);

**int** remain **=** **--**cowcount[PA2INDEX(pa)];

release(**&**kmem.lock);

**if** (remain **>** 0) {

*// 只有最后1个reference被删除时需要真正释放这个物理页*

**return**;

}

...省略...

}

**void** **\***

**kalloc**(**void**)

{

**struct** run **\***r;

acquire(**&**kmem.lock);

r **=** kmem.freelist;

**if**(r)

kmem.freelist **=** r**->**next;

release(**&**kmem.lock);

**if**(r) {

memset((**char** **\***)r, 5, PGSIZE); *// fill with junk*

**int** idx **=** PA2INDEX(r);

**if** (cowcount[idx] **!=** 0) {

panic("kalloc: cowcount[idx] != 0");

}

cowcount[idx] **=** 1; *// 新allocate的物理页的计数器为1*

}

**return** (**void\***)r;

}

*// helper函数*

**void** **adjustref**(uint64 pa, **int** num) {

**if** (pa **>=** PHYSTOP) {

panic("addref: pa too big");

}

acquire(**&**kmem.lock);

cowcount[PA2INDEX(pa)] **+=** num;

release(**&**kmem.lock);

}

如此一来,就可以在**fork()**时, 不再额外分配复制物理页, 而是让子进程和父进程共享只读页面.

**int**

**uvmcopy**(pagetable\_t old, pagetable\_t new, uint64 sz)

{

pte\_t **\***pte;

uint64 pa, i;

uint flags;

**for**(i **=** 0; i **<** sz; i **+=** PGSIZE){

**if**((pte **=** walk(old, i, 0)) **==** 0)

panic("uvmcopy: pte should exist");

**if**((**\***pte **&** PTE\_V) **==** 0)

panic("uvmcopy: page not present");

pa **=** PTE2PA(**\***pte);

**\***pte **&=** **~**PTE\_W; *// 变为只读页面, 不允许写. 一旦试图写, 会触发num=15的trap*

flags **=** PTE\_FLAGS(**\***pte);

**if**(mappages(new, i, PGSIZE, pa, flags) **!=** 0){

**goto** err;

}

adjustref(pa, 1); *// 增加计数器*

}

**return** 0;

err:

uvmunmap(new, 0, i **/** PGSIZE, 1);

**return** **-**1;

}

然后, 当一个进程想要对于一个只读的**COW**页面进行修改时, 需要把这一页复制一遍赋给这个进程. 单独写一个模块化函数来处理这一步.

**int**

**cowalloc**(pagetable\_t pagetable, uint64 va) {

**if** (va **>=** MAXVA) {

printf("cowalloc: exceeds MAXVA\n");

**return** **-**1;

}

pte\_t**\*** pte **=** walk(pagetable, va, 0);

**if** (pte **==** 0) {

panic("cowalloc: pte not exists");

}

**if** ((**\***pte **&** PTE\_V) **==** 0 **||** (**\***pte **&** PTE\_U) **==** 0) {

panic("cowalloc: pte permission err");

}

uint64 pa\_new **=** (uint64)kalloc();

**if** (pa\_new **==** 0) {

printf("cowalloc: kalloc fails\n");

**return** **-**1;

}

uint64 pa\_old **=** PTE2PA(**\***pte);

memmove((**void** **\***)pa\_new, (**const** **void** **\***)pa\_old, PGSIZE);

kfree((**void** **\***)pa\_old); *// 减少COW页面的reference count*

**\***pte **=** PA2PTE(pa\_new) **|** PTE\_FLAGS(**\***pte) **|** PTE\_W;

**return** 0;

}

当任何一个进程试图去写在这个页面时, 会触发代码号为**15**的缺页中断. 我们可以处理这个异常时为进程复制一页这个共享物理页, 并允许写权限.

**void**

**usertrap**(**void**)

{

...省略...

**if**(r\_scause() **==** 8){

*// system call*

**if**(p**->**killed)

exit(**-**1);

p**->**trapframe**->**epc **+=** 4;

intr\_on();

syscall();

} **else** **if** (r\_scause() **==** 15) {

**if** (cowalloc(p**->**pagetable, r\_stval()) **<** 0) {

p**->**killed **=** 1;

}

} **else** **if**((which\_dev **=** devintr()) **!=** 0){

*// ok*

} **else** {

printf("usertrap(): unexpected scause %p pid=%d\n", r\_scause(), p**->**pid);

printf(" sepc=%p stval=%p\n", r\_sepc(), r\_stval());

p**->**killed **=** 1;

}

...省略...

}

当复制一些数据到用户态的某一页时, 如果那一页是共享COW页, 要额外进行复制操作. 如何知道一个页是共享的COW页? 根据以上的实现, 如果一个页的**pte**里的**W**写flag被关闭了, 那就可以认为它是一个共享COW页.

**int**

**copyout**(pagetable\_t pagetable, uint64 dstva, **char** **\***src, uint64 len)

{

uint64 n, va0, pa0;

**while**(len **>** 0){

va0 **=** PGROUNDDOWN(dstva);

**if** (va0 **>=** MAXVA) {

printf("copyout: va exceeds MAXVA\n");

**return** **-**1;

}

pte\_t **\***pte **=** walk(pagetable, va0, 0);

**if** (pte **==** 0 **||** (**\***pte **&** PTE\_U) **==** 0 **||** (**\***pte **&** PTE\_V) **==** 0) {

printf("copyout: invalid pte\n");

**return** **-**1;

}

**if** ((**\***pte **&** PTE\_W) **==** 0) {

*// 写的目的地是COW共享页, 需要复制一份*

**if** (cowalloc(pagetable, va0) **<** 0) {

**return** **-**1;

}

}

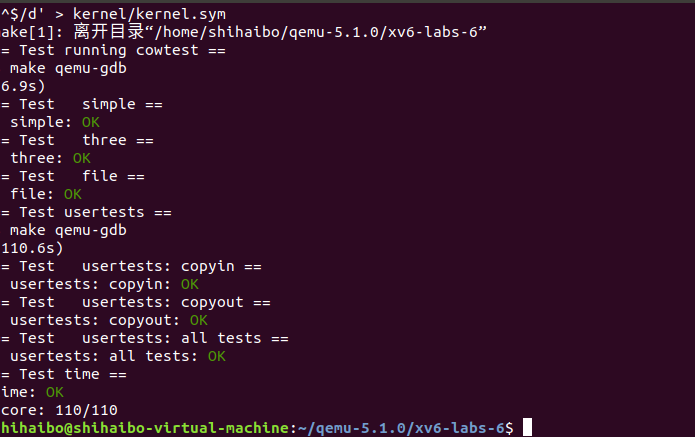
...省略...

}

**return** 0;

}

#### 6.3.2综合测试



### 6.4 实验中的问题和解决方法

1.内存共享错误：实验要求实现写时复制的fork机制，以便在父子进程之间共享物理内存页面。在处理内存共享时，可能会出现共享错误或内存访问冲突的情况。需要正确实现写时复制的逻辑，并确保父子进程能够正确地共享和访问内存。

2.页面复制错误：实验要求在父子进程之间延迟复制页面，只有在需要修改页面内容时才进行复制。在页面复制过程中，可能会出现页表映射错误、复制不完整或数据丢失的情况。需要正确实现页面复制的逻辑，并确保复制过程正确地更新页表和页面内容。

### 6.5 实验心得

实验 5 和本次实验都是关于内存分配策略的设计。本次实验实现了按需复制的 fork 策略，这也是实现了 fork 系统调用的操作系统中一般都会实现的优化方式。

此外，虽然在实验任务中并未提到，但是我观察到 xv6 系统实现了分页式的内存管理，这使得父子进程间的内存可以按照页面为单位进行拷贝，这也提高 了一定的内存有效数据比例、降低了一定的拷贝开销（相较于直接复制整个进程）。通过这次实验，我对操作系统中内存分配的方式和机制有了更深刻的体会，完成 COW 功能的过程也帮助我回顾了一块内存的分配、使用、销毁回收这个完整的生命周期。

## 七 Lab: Multithreading

### 7.1 实验目的

1. 熟悉多线程编程，理解进程切换方式和执行顺序。
2. 了解多核多线程对程序执行效率的影响。
3. 理解锁的含义，能够判断何时、何处应该加锁。
4. 回顾进程同步/互斥的模式并在程序中应用。

### 7.2 实验内容

1. 实现一个用户级进程的创建和切换。
2. 使用 UNIX pthread 线程库实现一个线程安全的哈希表。
3. 实现 barrier 函数。其作用是：当进程到达 barrier 函数调用时，会开始等待其他进程，当所有进程都到达 barrier 时，才停止等待。

### 7.3 实验过程

#### 7.3.1 switching between threads

1.首先把**kernel/switch.S**里用来保存寄存器状态的汇编代码原封不动的搬到**user/uthread\_switch.S**里.

\* save the old thread's registers,

\* restore the new thread's registers.

\*/

.globl thread\_switch

thread\_switch:

/\* YOUR CODE HERE \*/

sd ra, 0**(**a0**)**

sd sp, 8**(**a0**)**

sd s0, 16**(**a0**)**

sd s1, 24**(**a0**)**

sd s2, 32**(**a0**)**

sd s3, 40**(**a0**)**

sd s4, 48**(**a0**)**

sd s5, 56**(**a0**)**

sd s6, 64**(**a0**)**

sd s7, 72**(**a0**)**

sd s8, 80**(**a0**)**

sd s9, 88**(**a0**)**

sd s10, 96**(**a0**)**

sd s11, 104**(**a0**)**

ld ra, 0**(**a1**)**

ld sp, 8**(**a1**)**

ld s0, 16**(**a1**)**

ld s1, 24**(**a1**)**

ld s2, 32**(**a1**)**

ld s3, 40**(**a1**)**

ld s4, 48**(**a1**)**

ld s5, 56**(**a1**)**

ld s6, 64**(**a1**)**

ld s7, 72**(**a1**)**

ld s8, 80**(**a1**)**

ld s9, 88**(**a1**)**

ld s10, 96**(**a1**)**

ld s11, 104**(**a1**)**

ret /\* **return** to ra \*/

然后构造出**struct thread**用来保存每个用户态线程寄存器的**struct thread\_context**.

**struct** thread\_context {

uint64 ra;

uint64 sp;

uint64 s0;

uint64 s1;

uint64 s2;

uint64 s3;

uint64 s4;

uint64 s5;

uint64 s6;

uint64 s7;

uint64 s8;

uint64 s9;

uint64 s10;

uint64 s11;

};

**struct** **thread** {

**struct** thread\_context thread\_context; */\* register status \*/*

**char** stack[STACK\_SIZE]; */\* the thread's stack \*/*

**int** state; */\* FREE, RUNNING, RUNNABLE \*/*

};

需要一个小helper函数来初始化一个新线程的状态. 需要把它的第一次的返回地址**ra**设置成用户传进来的线程函数**func**的地址, 并且把这个线程的状态设置成***RUNNABLE***, 让调度器去跑它. 需要注意在**risc-v**里, 栈是由高地址向低地址增长的, 所以这个线程的最初stack pointer sp应该在栈的顶端.

**void** **clear\_thread**(**struct** **thread** **\***t, **void** (**\***func)()) {

memset((**void** **\***)**&**t**->**stack, 0, STACK\_SIZE);

memset((**void** **\***)**&**t**->**thread\_context, 0, **sizeof**(**struct** thread\_context));

t**->**state **=** RUNNABLE;

t**->**thread\_context.sp **=** (uint64) ((**char** **\***)**&**t**->**stack **+** STACK\_SIZE);

t**->**thread\_context.ra **=** (uint64) func;

}

然后把缺失的两个 **/\**YOUR CODE HERE \**/** 的地方填补上即可.

**void**

**thread\_schedule**(**void**)

{

**struct** **thread** **\***t, **\***next\_thread;

**if** (current\_thread **!=** next\_thread) {

next\_thread**->**state **=** RUNNING;

t **=** current\_thread;

current\_thread **=** next\_thread;

thread\_switch((uint64) t, (uint64) current\_thread);

} **else**

next\_thread **=** 0;

}

**void**

**thread\_create**(**void** (**\***func)())

{

**struct** **thread** **\***t;

**for** (t **=** all\_thread; t **<** all\_thread **+** MAX\_THREAD; t**++**) {

**if** (t**->**state **==** FREE) **break**;

}

clear\_thread(t, func);

}

#### 7.3.2 Using thread

直接为每个**bucket**配置一个**pthread\_mutex**锁来保证只有一个线程可以读这个**bucket**里的所有数据进行读和写的操作。

pthread\_mutex\_t locks[NBUCKET]; *// one lock per bucket*

**void** **init\_locks**() {

**for** (**int** i **=** 0; i **<** NBUCKET; i**++**) {

pthread\_mutex\_init(**&**locks[i], NULL);

}

}

**static**

**void** **put**(**int** key, **int** value)

{

**int** i **=** key **%** NBUCKET;

pthread\_mutex\_lock(**&**locks[i]);

**struct** entry **\***e **=** 0;

**for** (e **=** table[i]; e **!=** 0; e **=** e**->**next) {

**if** (e**->**key **==** key)

**break**;

}

**if**(e){

e**->**value **=** value;

} **else** {

insert(key, value, **&**table[i], table[i]);

}

pthread\_mutex\_unlock(**&**locks[i]);

}

**static** **struct** entry**\***

**get**(**int** key)

{

**int** i **=** key **%** NBUCKET;

pthread\_mutex\_lock(**&**locks[i]);

**struct** entry **\***e **=** 0;

**for** (e **=** table[i]; e **!=** 0; e **=** e**->**next) {

**if** (e**->**key **==** key) **break**;

}

pthread\_mutex\_unlock(**&**locks[i]);

**return** e;

}

#### 7.3.3 barrier

这个实验需要一个**barrier**, 每个线程都会呼叫**barrier()**函数, 将会卡在里面直到每个需要synchronize的线程都进入这个函数. 需要使用**conditional variable**来实现等待和唤醒的功能.

但在实现的时候需要注意一点是, 在所以thread离开上一轮之前, **bstate.nthread**不能被下一轮的第一个人设置为**0**从而形成竞态, 使得一些上一轮沉睡等待的thread卡死在上一轮中. 在此处使用C新特性**thread local**来实现**sense reversal**.

**static** \_\_thread **int** thread\_flag **=** 0;

**struct** barrier {

pthread\_mutex\_t barrier\_mutex;

pthread\_cond\_t barrier\_cond;

**int** nthread;

**int** round;

**int** flag;

} bstate;

**static** **void**

**barrier\_init**(**void**)

{

assert(pthread\_mutex\_init(**&**bstate.barrier\_mutex, NULL) **==** 0);

assert(pthread\_cond\_init(**&**bstate.barrier\_cond, NULL) **==** 0);

bstate.round **=** 0;

bstate.nthread **=** 0;

bstate.flag **=** 0;

}

**static** **void**

**barrier**()

{

pthread\_mutex\_lock(**&**bstate.barrier\_mutex);

thread\_flag**=** **!**thread\_flag;

**while** (thread\_flag **==** bstate.flag) {

pthread\_cond\_wait(**&**bstate.barrier\_cond, **&**bstate.barrier\_mutex);

}

**int** arrived **=** **++**bstate.nthread;

**if** (arrived **==** nthread) {

bstate.round**++**;

bstate.flag **=** **!**bstate.flag;

bstate.nthread **=** 0;

pthread\_cond\_broadcast(**&**bstate.barrier\_cond);

} **else** {

pthread\_cond\_wait(**&**bstate.barrier\_cond, **&**bstate.barrier\_mutex);

}

pthread\_mutex\_unlock(**&**bstate.barrier\_mutex);

}

#### 7.3.4综合测试



### 7.4 实验中的问题和解决方法

1.竞争条件：多个线程同时访问和修改共享资源时，出现竞争条件，导致结果不一致或意外行为。由于缺乏适当的同步机制，如互斥锁或信号量。解决此问题的方法是使用同步机制来保护共享资源，并确保只有一个线程可以访问资源。

2.死锁：当多个线程相互等待对方所持有的资源时，可能发生死锁，导致线程无法继续执行。这可能是由于资源的顺序获取问题或互斥锁的使用不正确。要解决死锁问题，需要仔细设计同步策略，并确保资源的正确释放和线程的协调。

### 7.5 实验心得

本实验是关于多线程编程的。之前并没有编写过多线程的程序，只知道多线程程序可以运用多个处理器，可以达到提高速度的效果。在操作系统课程中介绍了锁的概念和几个模型，并讨论了多线程存在的死锁问题。在实验中，实现了进程切换、运用锁来完成线程间的互斥（多线程哈希表）和运用锁和条件信号量来完成多线程之间的同步（barrier），三个小实验加深了我对于多线程的理解，提供了练习多线程编程的机会。

## 八 Lab: Lock

### 8.1 实验目的

1. 理解锁竞争，包括锁竞争产生的原因以及解决方法。
2. 理解通过增加锁的数量来降低锁竞争的原理，并应用到程序中。
3. 加深对于使用锁来达到进程互斥方法的掌握程度。

### 实验内容

1. 重新设计系统管理内存的方式，实现每个 CPU 管理一个空闲链表和一个锁。
2. 修改系统的 IO 缓冲区，通过散列的方式完成对缓冲区和锁的分割。

### 8.3 实验过程

#### 8.3.1 Memory allocator

首先改写数据结构, 改为每**cpu核**一个freelist.

**struct** {

**struct** spinlock lock;

**struct** run **\***freelist;

} kmem[NCPU];

**void**

**kinit**()

{

**for** (**int** i **=** 0; i **<** NCPU; i**++**) {

**char** name[9] **=** {0};

snprintf(name, 8, "kmem-%d", i);

initlock(**&**kmem[i].lock, name);

}

freerange(end, (**void\***)PHYSTOP);

}

在**kalloc**和**kfree**的时候要注意暂时先关闭interrupt中断, 免得在读取到当前**cpu id**是**1**后, 这一段代码被放到**cpu 2**上去跑, 造成内存不均衡.

**void**

**kfree**(**void** **\***pa)

{

**struct** run **\***r;

**if**(((uint64)pa **%** PGSIZE) **!=** 0 **||** (**char\***)pa **<** end **||** (uint64)pa **>=** PHYSTOP)

panic("kfree");

push\_off();

**int** cpu **=** cpuid();

memset(pa, 1, PGSIZE);

r **=** (**struct** run**\***)pa;

acquire(**&**kmem[cpu].lock);

r**->**next **=** kmem[cpu].freelist;

kmem[cpu].freelist **=** r;

release(**&**kmem[cpu].lock);

pop\_off();

}

写一个helper函数, 以**round-robin**的形式去偷取**邻居cpu**的空余内存页, 一次只偷取一页. 其实可以想到的是, 如果想达成一个更高效的运行, 即减少偷取的次数, 最好是一次偷取比如邻居一半的空余内存页. 这需维持额外一个**int**结构来记住每个freelist现在有多长.

**void** **\***

**ksteal**(**int** cpu) {

**struct** run **\***r;

**for** (**int** i **=** 1; i **<** NCPU; i**++**) {

**int** next\_cpu **=** (cpu **+** i) **%** NCPU;

acquire(**&**kmem[next\_cpu].lock);

r **=** kmem[next\_cpu].freelist;

**if** (r) {

kmem[next\_cpu].freelist **=** r**->**next;

}

release(**&**kmem[next\_cpu].lock);

**if** (r) {

**break**;

}

}

**return** r;

}

**void** **\***

**kalloc**(**void**)

{

**struct** run **\***r;

push\_off();

**int** cpu **=** cpuid();

acquire(**&**kmem[cpu].lock);

r **=** kmem[cpu].freelist;

**if** (r) {

kmem[cpu].freelist **=** r**->**next;

}

release(**&**kmem[cpu].lock);

**if** (r **==** 0) {

r **=** ksteal(cpu);

}

**if**(r)

memset((**char\***)r, 5, PGSIZE);

pop\_off();

**return** (**void\***)r;

}

#### 8.3.2 Buffer cash

首先改写一下数据结构, 并把两个很简单的小功能**bpin**和**bunpin**修改一下, 然后在**binit**里把每个锁都初始化一下.

**extern** uint ticks;

**struct** {

**struct** spinlock lock;

**struct** buf buf[BUFFERSIZE];

} bcachebucket[BUCKETSIZE];

**int**

**hash**(uint blockno)

{

**return** blockno **%** BUCKETSIZE;

}

**void**

**bpin**(**struct** buf **\***b) {

**int** bucket **=** hash(b**->**blockno);

acquire(**&**bcachebucket[bucket].lock);

b**->**refcnt**++**;

release(**&**bcachebucket[bucket].lock);

}

**void**

**bunpin**(**struct** buf **\***b) {

**int** bucket **=** hash(b**->**blockno);

acquire(**&**bcachebucket[bucket].lock);

b**->**refcnt**--**;

release(**&**bcachebucket[bucket].lock);

}

**void**

**binit**(**void**)

{

**for** (**int** i **=** 0; i **<** BUCKETSIZE; i**++**) {

initlock(**&**bcachebucket[i].lock, "bcachebucket");

**for** (**int** j **=** 0; j **<** BUFFERSIZE; j**++**) {

initsleeplock(**&**bcachebucket[i].buf[j].lock, "buffer");

}

}

}

最重点的**bget**函数: 根据所需要的**blockno**, 计算出对应哪个bucket后, 拿锁进行查找. 如果没能找到对应的buffer cache block, 则就在当前bucket里试图寻找一个空闲的来分配.

**static** **struct** buf**\***

**bget**(uint dev, uint blockno)

{

**struct** buf **\***b;

**int** bucket **=** hash(blockno);

acquire(**&**bcachebucket[bucket].lock);

**for** (**int** i **=** 0; i **<** BUFFERSIZE; i**++**) {

b **=** **&**bcachebucket[bucket].buf[i];

**if** (b**->**dev **==** dev **&&** b**->**blockno **==** blockno) {

b**->**refcnt**++**;

b**->**lastuse **=** ticks;

release(**&**bcachebucket[bucket].lock);

acquiresleep(**&**b**->**lock);

*// --- end of critical session*

**return** b;

}

}

**int** least\_idx **=** **-**1;

**for** (**int** i **=** 0; i **<** BUFFERSIZE; i**++**) {

b **=** **&**bcachebucket[bucket].buf[i];

**if**(b**->**refcnt **==** 0 **&&** b**->**lastuse **<** least) {

least **=** b**->**lastuse;

least\_idx **=** i;

}

}

**if** (least\_idx **==** **-**1) {

panic("bget: no unused buffer for recycle");

}

b **=** **&**bcachebucket[bucket].buf[least\_idx];

b**->**dev **=** dev;

b**->**blockno **=** blockno;

b**->**lastuse **=** ticks;

b**->**valid **=** 0;

b**->**refcnt **=** 1;

release(**&**bcachebucket[bucket].lock);

acquiresleep(**&**b**->**lock);

**return** b;

}

**void**

**brelse**(**struct** buf **\***b)

{

**if**(**!**holdingsleep(**&**b**->**lock))

panic("brelse");

**int** bucket **=** hash(b**->**blockno);

acquire(**&**bcachebucket[bucket].lock);

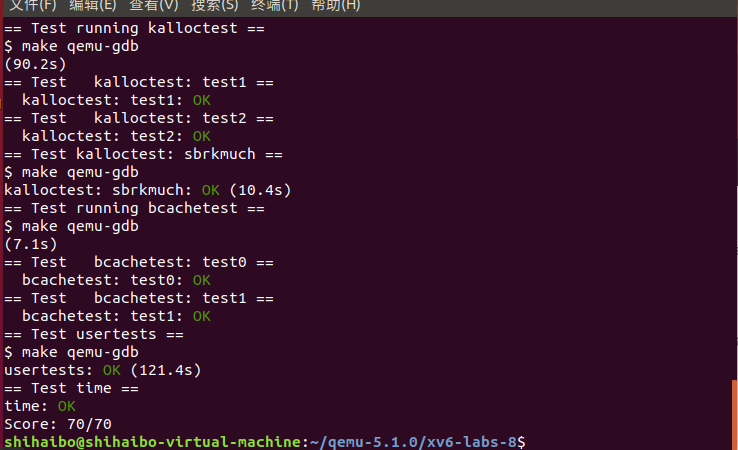
b**->**refcnt**--**;

release(**&**bcachebucket[bucket].lock);

releasesleep(**&**b**->**lock);

}

#### 8.3.3综合测试



### 8.4 实验中的问题和解决方法

1.死锁：实验要求实现锁机制来确保共享资源的正确同步。在处理锁的获取和释放时，可能会出现死锁的情况，即多个进程或线程互相等待对方释放锁而无法继续执行。需要正确地设计和实现锁的获取和释放顺序，并避免循环依赖。

2.竞争条件：实验可能要求你处理并发访问共享资源的竞争条件。在没有合适的同步措施时，多个进程或线程可能同时修改共享资源，导致数据不一致或错误的结果。需要使用适当的锁机制来保护共享资源，并确保只有一个进程或线程能够修改该资源。

### 8.5 实验心得

1.程序来更好地使用处理器资源，而分时系统通过多个进程的并发运行可以让多个程序看起来像同时在执行。在处理多线程或多进程中资源抢占而导致的锁竞争现象，常用的做法是通过分割资源并且分别加锁。锁的数量越多，单个锁上的冲突就越少（但是也要考虑锁的数量增多的负面影响）。

2.我在两个实验中分别体会了控制进程的同步/互斥（实验 7）和分割资源来使锁竞争得到缓解、提高资源并行利用率。实验同时还涉及到了关中断、散列、资源分配等其他方面的细节，关联了我之前课程中学习的知识。从功能性角度来说，一个操作系统只需要拥有运行程序的能力，与用户交互的系统也最多需要增加并发运行这一个功能限制。而在实验中通过对资源的划分实现了提高系统效率和资源利用率这些非功能性需求，在一个面向现实世界的操作系统中，非功能性需求一样有重要地位。

## 九 Lab: File system

### 9.1 实验目的

1. 理解 UNIX 系统以及 xv6 系统组织文件的方式，包括 inode 数据结构以及地址的映射关系。
2. 掌握通过增加索引级数的文索引扩展方式。
3. 理解文件映射和打开的方式，了解 xv6 系统的文件系统层级架构。
4. 理解硬链接和软链接的实现方式。

### 9.2 实验内容

1. 修改文件系统使其支持更大的文件存储。具体做法是实现二级间接的索引节点（inode）。
2. 增加一个通过符号链接的系统调用。将目标符号与系统路径链接，使对符号文件的操作都同步到对相应路径文件的操作。

### 9.3 实验过程

#### 9.3.1 Large files

首先改写一下宏定义.

**struct** dinode {

**short** type; *// File type*

**short** major; *// Major device number (T\_DEVICE only)*

**short** minor; *// Minor device number (T\_DEVICE only)*

**short** nlink; *// Number of links to inode in file system*

uint size; *// Size of file (bytes)*

uint addrs[NDIRECT**+**2]; *// >>> 这里由+1变成+2 <<<*

};

**struct** inode {

uint dev; *// Device number*

uint inum; *// Inode number*

**int** ref; *// Reference count*

**struct** sleeplock lock; *// protects everything below here*

**int** valid; *// inode has been read from disk?*

**short** type; *// copy of disk inode*

**short** major;

**short** minor;

**short** nlink;

uint size;

uint addrs[NDIRECT**+**2]; *// >>> 这里由+1变成+2 <<<*

};

然后在**bmap**寻找block的函数里加上双层间接映射的逻辑, 同时在**itrunc**里把对应的双层连接也清除掉.

**static** uint

**bmap**(**struct** inode **\***ip, uint bn)

{

uint addr, **\***a, **\***b;

**struct** buf **\***inbp, **\***ininbp;

... 直接映射层 和之前代码一样 ...

bn **-=** NDIRECT;

... 单层间接映射 和之前代码一样 ...

bn **-=** NINDIRECT; *// 记得减掉offset*

**if** (bn **<** NININDIRECT)

**if** ((addr **=** ip**->**addrs[NDIRECT**+**1]) **==** 0)

ip**->**addrs[NDIRECT**+**1] **=** addr **=** balloc(ip**->**dev);

inbp **=** bread(ip**->**dev, addr);

a **=** (uint**\***)inbp**->**data;

**if** ((addr **=** a[bn**/**NINDIRECT]) **==** 0) {

a[bn**/**NINDIRECT] **=** addr **=** balloc(ip**->**dev);

log\_write(inbp);

}

brelse(inbp);

ininbp **=** bread(ip**->**dev, addr);

b **=** (uint**\***)ininbp**->**data;

**if** ((addr **=** b[bn **%** NINDIRECT]) **==** 0) { *// 取余数*

b[bn **%** NINDIRECT] **=** addr **=** balloc(ip**->**dev);

log\_write(ininbp);

}

brelse(ininbp);

**return** addr;

}

panic("bmap: out of range");

}

**void**

itrunc(**struct** inode **\***ip)

{

**int** i, j, k;

**struct** buf **\***bp, **\***inbp;

uint **\***a, **\***b;

... 直接映射层和单层间接映射层 和之前代码一样 ...

**if**(ip**->**addrs[NDIRECT**+**1]) {

bp **=** bread(ip**->**dev, ip**->**addrs[NDIRECT**+**1]);

a **=** (uint**\***)bp**->**data;

**for** (j **=** 0; j **<** NINDIRECT; j**++**) {

**if** (a[j]) {

inbp **=** bread(ip**->**dev, a[j]);

b **=** (uint**\***)inbp**->**data;

**for** (k **=** 0; k **<** NINDIRECT; k**++**) {

**if** (b[k])

bfree(ip**->**dev, b[k]);

}

brelse(inbp);

bfree(ip**->**dev, a[j]);

}

}

brelse(bp);

bfree(ip**->**dev, ip**->**addrs[NDIRECT**+**1]);

ip**->**addrs[NDIRECT**+**1] **=** 0;

}

ip**->**size **=** 0;

iupdate(ip);

}

#### 9.3.2 Symbolic links

首先按常规操作, 把一个新的系统调用**sys\_symlink**的代号, 跳转函数等等设立好. 然后直接开始看**sys\_symlink**的实现: 把其对应的**inode**的type设为**T\_SYMLINK**, 然后在其data block的**[0, MAXPATH]**的范围里写上所要链接的**target path**的路径.

**int** **sys\_symlink**(**char** **\***target, **char** **\***path) {

**char** kpath[MAXPATH], ktarget[MAXPATH];

memset(kpath, 0, MAXPATH);

memset(ktarget, 0, MAXPATH);

**struct** inode **\***ip;

**int** n, r;

**if**((n **=** argstr(0, ktarget, MAXPATH)) **<** 0)

**return** **-**1;

**if** ((n **=** argstr(1, kpath, MAXPATH)) **<** 0)

**return** **-**1;

**int** ret **=** 0;

begin\_op();

**if**((ip **=** namei(kpath)) **!=** 0){

*// symlink already exists*

ret **=** **-**1;

**goto** final;

}

ip **=** create(kpath, T\_SYMLINK, 0, 0);

**if**(ip **==** 0){

ret **=** **-**1;

**goto** final;

}

**if** ((r **=** writei(ip, 0, (uint64)ktarget, 0, MAXPATH)) **<** 0)

ret **=** **-**1;

iunlockput(ip);

final:

end\_op();

**return** ret;

}

然后在**sys\_open**这个函数里, 如果传入的**path**是一个软链接的话, 需要为用户递归去"***寻址***", 直到找到第一个不是软链接的**path**

uint64

**sys\_open**(**void**)

{

... 省略, 和之前源码一样 ...

**if**(omode **&** O\_CREATE){

... 省略, 和之前源码一样 ...

}

*// 是软链接且O\_NOFOLLOW没被设立起来*

**int** depth **=** 0;

**while** (ip**->**type **==** T\_SYMLINK **&&** **!**(omode **&** O\_NOFOLLOW)) {

**char** ktarget[MAXPATH];

memset(ktarget, 0, MAXPATH);

*// 从软链接的inode的[0, MAXPATH]读出它所对应的target path*

**if** ((r **=** readi(ip, 0, (uint64)ktarget, 0, MAXPATH)) **<** 0) {

iunlockput(ip);

end\_op();

**return** **-**1;

}

iunlockput(ip);

**if**((ip **=** namei(ktarget)) **==** 0){ *// target path 不存在*

end\_op();

**return** **-**1;

}

ilock(ip);

depth**++**;

**if** (depth **>** 10) {

*// maybe form a cycle 默认死循环*

iunlockput(ip);

end\_op();

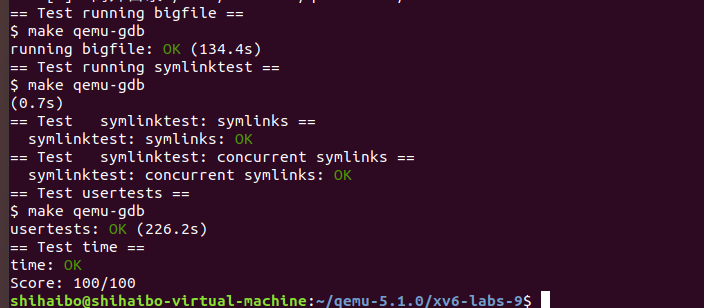
**return** **-**1;

}

}

}

#### 9.3.3综合测试



### 9.4 实验中的问题和解决方法

1.文件元数据错误：实验可能要求你维护文件的元数据信息，如文件名、文件大小、时间戳等。在处理文件元数据时，可能会出现错误的元数据更新、读取或不一致的情况。要解决这些问题，需要正确地维护和更新文件的元数据，并确保文件操作与元数据的一致性。

2.并发访问错误：实验可能要求你处理多个进程或线程对文件系统的并发访问。在没有合适的同步机制时，多个进程或线程可能同时修改文件系统的结构，导致数据不一致或文件损坏。要解决这些问题，需要使用合适的同步机制（如锁或信号量）来保护文件系统的并发访问，并确保只有一个进程或线程能够修改文件系统的结构。

### 9.5 实验心得

本次试验是关于 xv6 系统文件系统的，拓展了可以索引的文件大小，也创建了符号链接这种软链接方式。

1.第一个部分中的拓展文件大小是一个典型的程序数据结构设计思想，可以看作是对于一个高度为 1 的树，将其某一些节点拓展成指向相似的节点的非叶子结点，从而扩展这个数的容量（使其高度增加）。这种思想被运用在可拓展地址数的操作数设计（计算机组成原理课程中），以及多级页表的设计（操作系统课程中）。手动实现这一功能的过程十分有趣。

2.第二个部分是实现系统文件软链接相关的内容。这个部分在课程内要求不高， 在实现的过程中最大的困难在于理解“软链接也是一种文件”这一概念。这个实 验让我加深了对电脑上那些左下角有箭头符号的文件本质是什么这一问题理解。

## 十 Lab: Mmaps

### 10.1 实验目的

1. 了解使用虚拟内存的原因和好处。
2. 理解创建和释放文件映射的方式。
3. 梳理文件内存映射的生命周期。

### 10.2 实验内容

添加 mmap 与 mumap 两个系统调用。前者将一个文件内存映射到进程的地址空间，后者取消已有地址空间的映射。本实验的实验内容只有一条，但是难度被标为困难（hard），因此我还是将实验步骤细分，这样更利于实验的理解和报告的组织。细分后的实验分为三步：

1. 理解虚拟地址映射和访问的方式，了解 mmap 系统调用的实际工作流程。
2. 实现 mmap 系统调用。
3. 实现 munmap 系统调用。
4. 修改其他相关的函数。

### 10.3 实验过程

#### 10.3.1 mmap

按照实验要求, 为每个struct proc增加一个长度为16的vma表

**struct** vma {

**int** valid;

uint64 addr;

**int** length;

**int** prot;

**int** flags;

**int** fd;

**int** offset;

**struct** file**\*** f;

};

**struct** proc {

...省略...

**char** name[16];

**struct** vma vmas[NVMA];

};

首先把两个新的系统调用**mmap**和**munmap**的一系列函数签名, 调用代码, 跳转指针等常规操作都做好. 为了效率, 并不直接分配物理页, 因为用户或许只会读到一个很大的文件的一小部分. 但是确实通过增加**p->sz**这个参数, 为整个文件大小在用户态的地址空间里预留了位置.

uint64

**sys\_mmap**(**void**) {

uint64 failure **=** (uint64)((**char** **\***) **-**1);

**struct** proc**\*** p **=** myproc();

uint64 addr;

**int** length, prot, flags, fd, offset;

**struct** file**\*** f;

**if** (argaddr(0, **&**addr) **<** 0 **||** argint(1, **&**length) **<** 0 **||** argint(2, **&**prot) **<** 0

**||** argint(3, **&**flags) **<** 0 **||** argfd(4, **&**fd, **&**f) **<** 0 **||** argint(5, **&**offset) **<** 0)

**return** failure;

length **=** PGROUNDUP(length);

**if** (MAXVA **-** length **<** p**->**sz)

**return** failure;

**if** (**!**f**->**readable **&&** (prot **&** PROT\_READ))

**return** failure;

**if** (**!**f**->**writable **&&** (prot **&** PROT\_WRITE) **&&** (flags **==** MAP\_SHARED))

**return** failure;

**for** (**int** i **=** 0; i **<** NVMA; i**++**) {

**struct** vma**\*** vma **=** **&**p**->**vmas[i];

**if** (vma**->**valid **==** 0) {

vma**->**valid **=** 1;

vma**->**addr **=** p**->**sz;

p**->**sz **+=** length;

vma**->**length **=** length;

vma**->**prot **=** prot;

vma**->**flags **=** flags;

vma**->**fd **=** fd;

vma**->**f **=** f;

filedup(f);

vma**->**offset **=** offset;

**return** vma**->**addr;

}

}

**return** failure;

}

在这样写完**mmap**后, 当用户试图去访问**mmap**所返回的地址时, 由于没有分配物理页, 将会触发**缺页中断**. 这个时候就需要在**usertrap**里把对应offset的文件内容读到一个新分配的物理页中, 并把这个物理页加入这个进程的虚拟内存映射表里.

**# kernel/trap.c**

**void**

**usertrap**(**void**)

{

...省略...

**if**(r\_scause() **==** 8){

...省略...

} **else** **if**(r\_scause() **==** 13 **||** r\_scause() **==** 15) { *// 读或写造成的缺页中断*

uint64 va **=** r\_stval();

**struct** proc**\*** p **=** myproc();

**if** (va **>** MAXVA **||** va **>** p**->**sz) {

*// sanity check安全检查*

p**->**killed **=** 1;

} **else** {

**int** found **=** 0;

**for** (**int** i **=** 0; i **<** NVMA; i**++**) {

**struct** vma**\*** vma **=** **&**p**->**vmas[i];

**if** (vma**->**valid **&&** va **>=** vma**->**addr **&&** va **<** vma**->**addr**+**vma**->**length) {

va **=** PGROUNDDOWN(va);

uint64 pa **=** (uint64)kalloc();

**if** (pa **==** 0) {

**break**;

}

memset((**void** **\***)pa, 0, PGSIZE);

ilock(vma**->**f**->**ip);

**if**(readi(vma**->**f**->**ip, 0, pa, vma**->**offset **+** va **-** vma**->**addr, PGSIZE) **<** 0) {

iunlock(vma**->**f**->**ip);

**break**;

}

iunlock(vma**->**f**->**ip);

**int** perm **=** PTE\_U; *// 权限设置*

**if** (vma**->**prot **&** PROT\_READ)

perm **|=** PTE\_R;

**if** (vma**->**prot **&** PROT\_WRITE)

perm **|=** PTE\_W;

**if** (vma**->**prot **&** PROT\_EXEC)

perm **|=** PTE\_X;

**if** (mappages(p**->**pagetable, va, PGSIZE, pa, perm) **<** 0) {

kfree((**void\***)pa);

**break**;

}

found **=** 1;

**break**;

}

}

**if** (**!**found)

p**->**killed **=** 1;

}

}

...省略...

}

然后, 在**munmap**时, 需要把分配的物理页释放掉, 而且如果flag是**MAP\_SHARED**, 直接把**unmap**的区域无脑复写回文件中

uint64

**sys\_munmap**(**void**) {

uint64 addr;

**int** length;

**if** (argaddr(0, **&**addr) **<** 0 **||** argint(1, **&**length) **<** 0)

**return** **-**1;

**struct** proc **\***p **=** myproc();

**struct** vma**\*** vma **=** 0;

**int** idx **=** **-**1;

**for** (**int** i **=** 0; i **<** NVMA; i**++**) {

**if** (p**->**vmas[i].valid **&&** addr **>=** p**->**vmas[i].addr **&&** addr **<=** p**->**vmas[i].addr **+** p**->**vmas[i].length) {

idx **=** i;

vma **=** **&**p**->**vmas[i];

**break**;

}

}

**if** (idx **==** **-**1)

**return** **-**1;

addr **=** PGROUNDDOWN(addr);

length **=** PGROUNDUP(length);

**if** (vma**->**flags **&** MAP\_SHARED) {

**if** (filewrite(vma**->**f, addr, length) **<** 0) {

printf("munmap: filewrite < 0\n");

}

}

uvmunmap(p**->**pagetable, addr, length**/**PGSIZE, 1);

*// change the mmap parameter*

**if** (addr **==** vma**->**addr **&&** length **==** vma**->**length) {

*// fully unmapped 完全释放*

fileclose(vma**->**f);

vma**->**valid **=** 0;

} **else** **if** (addr **==** vma**->**addr) {

*// cover the beginning 释放区域包括头部*

vma**->**addr **+=** length;

vma**->**length **-=** length;

vma**->**offset **+=** length;

} **else** **if** ((addr **+** length) **==** (vma**->**addr **+** vma**->**length)) {

*// cover the end 释放区域包括尾部*

vma**->**length **-=** length;

} **else** {

panic("munmap neither cover beginning or end of mapped region");

}

**return** 0;

}

最后, 在**exit**时也要把**mmap**的区域释放掉. 在**fork**时需要拷贝**vma**表, 并增加一次文件**f**的引用数量

**void**

**exit**(**int** status)

{

**for** (**int** i **=** 0; i **<** NVMA; i**++**) {

**if** (p**->**vmas[i].valid) {

**if** (p**->**vmas[i].flags **&** MAP\_SHARED) {

filewrite(p**->**vmas[i].f, p**->**vmas[i].addr, p**->**vmas[i].length);

}

fileclose(p**->**vmas[i].f);

uvmunmap(p**->**pagetable, p**->**vmas[i].addr, p**->**vmas[i].length **/** PGSIZE, 1);

p**->**vmas[i].valid **=** 0;

}

}

begin\_op();

}

**int**

**fork**(**void**)

{

**for** (**int** i **=** 0; i **<** NVMA; i**++**) {

np**->**vmas[i].valid **=** 0;

**if** (p**->**vmas[i].valid) { *// 复制vma entry*

memmove(**&**np**->**vmas[i], **&**p**->**vmas[i], **sizeof**(**struct** vma));

filedup(p**->**vmas[i].f); *// 增加引用次数*

}

}

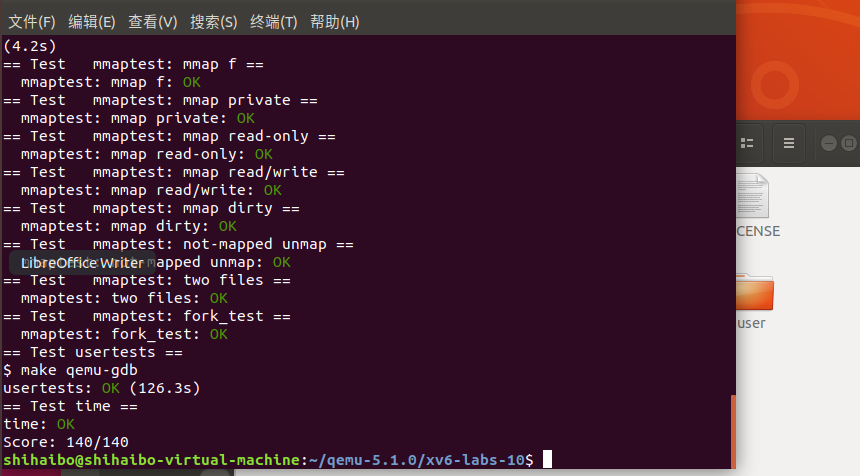
np**->**state **=** RUNNABLE;

release(**&**np**->**lock);

**return** pid;

}

#### 10.3.2综合测试



### 10.4 实验中的问题和解决方法

1.文件打开、读取和写入错误：在实验中，你将实现文件的打开、读取和写入操作。在处理这些操作时，可能会出现文件句柄管理错误、读取/写入位置计算错误或者读取/写入字节数不正确的情况。要解决这个问题，需要确保正确实现文件句柄的管理，正确计算读取/写入位置，并检查读取/写入操作的字节数。

2.目录操作错误：实验可能要求你实现目录的创建、删除和搜索等操作。在处理目录操作时，可能会出现目录结构损坏、目录项操作错误或者目录遍历错误的情况。要解决这个问题，需要正确实现目录操作的逻辑，并确保目录结构的一致性和正确性。

### 10.5 实验心得

1.本次实验也是关于系统与文件交互的内容，通过实验，实现了一个磁盘文件到虚拟内存地址的映射方式。这次实验的内容较多，覆盖了 xv6 系统中虚拟内存、文件地址、懒加载等知识点。在实验的提示中详细地列举了需要完成的对源代码的修改和添加，但是数目很多，稍有遗漏就会产生难以排查的 bug。

2.实验中需要调用的系统函数众多，这也是在实现两个系统调用时比较难以理解的部分。例如在映射一个页面时需要调用 mappages，而在取消映射关系时则会调用 uvmunmap 函数。此外，该实验涉及到了许多关于文件的操作，都让阅读了很多相关的源码和 xv6-book 相应的章节。

3.这个实验让我印象最深刻的地方在于对于页面懒加载的实现以及vma 数据结构的设计。懒加载这种方式在实验 5和实验 6中都有涉及，中心思想是不能一次性加载、拷贝所有内存，因为大多时候用不到。这种方式的泛用性印证了懒加载思想的重要地位。而在设计 vma 数据结构时，除了记录地址、长度、权限、文件等描述中提醒的字段之外，还记录了该内存映射地址是否正在使用、文件的共享标记这两个必要的字段。提出这两个字段是开始编写 mmap 系统调用时发现实现中有缺漏或是困难，才发现有字段缺失。

## 十一 Lab: Network driver

### 11.1 实验目的

1. 理解网络驱动程序的工作原理：通过实践，学生可以深入了解网络驱动程序如何与硬件设备进行交互，包括设备初始化、数据包接收和发送、中断处理等。
2. 掌握网络数据包的处理过程：学生将学习如何处理不同类型的网络数据包，实现数据包的封装、解析、路由选择等功能。
3. 熟悉操作系统内核的网络子系统：通过实现网络驱动程序，学生将熟悉操作系统内核中与网络相关的子系统，如网络栈、套接字接口等。
4. 锻炼代码调试和故障排除能力：网络驱动程序的实现可能涉及复杂的代码逻辑和底层操作，学生将锻炼调试和故障排除的能力，以解决实际问题。

### 11.2 实验内容

1. 设备初始化：需要编写代码来初始化网络设备，包括设置设备参数、启动设备等操作。
2. 中断处理：需要处理网络设备产生的中断，以接收和处理传入的数据包或其他事件。
3. 数据包接收：需要实现网络驱动程序来接收传入的数据包，并将其传递给操作系统的网络协议栈进行进一步处理。
4. 数据包发送：需要实现网络驱动程序来发送数据包，将数据从操作系统发送到网络设备，以便通过网络发送出去。
5. 缓冲区管理：需要设计和管理缓冲区，以存储接收到的数据包或待发送的数据。
6. 错误处理和故障排除：需要实现适当的错误处理和故障排除机制，以检测和应对可能发生的问题，并确保网络驱动程序的稳定性和可靠性。

### 11.3 实验过程

#### 11.3.1 network

在本次实验中, 只需要关心**kernel/e1000.c**里的两个留空函数的实现.

**e1000\_transmit**是给了一个新的**packet**, 我们需要在ring里找到下一个空余位置, 然后把它放进去等待传输.

**int**

**e1000\_transmit**(**struct** mbuf **\***m)

{

acquire(**&**e1000\_lock);

**int** idx **=** regs[E1000\_TDT];

**if** ((tx\_ring[idx].status **&** E1000\_TXD\_STAT\_DD) **==** 0) {

release(**&**e1000\_lock);

**return** **-**1;

}

**if** (tx\_mbufs[idx])

mbuffree(tx\_mbufs[idx]);

tx\_mbufs[idx] **=** m;

tx\_ring[idx].length **=** m**->**len;

tx\_ring[idx].addr **=** (uint64) m**->**head;

tx\_ring[idx].cmd **=** E1000\_TXD\_CMD\_RS **|** E1000\_TXD\_CMD\_EOP;

regs[E1000\_TDT] **=** (idx **+** 1) **%** TX\_RING\_SIZE;

release(**&**e1000\_lock);

**return** 0;

}

**e1000\_recv**是需要遍历这个ring, 把所有新到来的**packet**交由网络上层的协议/应用去处理.

**static** **void**

**e1000\_recv**(**void**)

{

**while** (1) {

**int** idx **=** (regs[E1000\_RDT] **+** 1) **%** RX\_RING\_SIZE;

**if** ((rx\_ring[idx].status **&** E1000\_RXD\_STAT\_DD) **==** 0) {

*// 没有新包了*

**return**;

}

rx\_mbufs[idx]**->**len **=** rx\_ring[idx].length;

*// 向上层network stack传输*

net\_rx(rx\_mbufs[idx]);

*// 把这个下标清空 放置一个空包*

rx\_mbufs[idx] **=** mbufalloc(0);

rx\_ring[idx].status **=** 0;

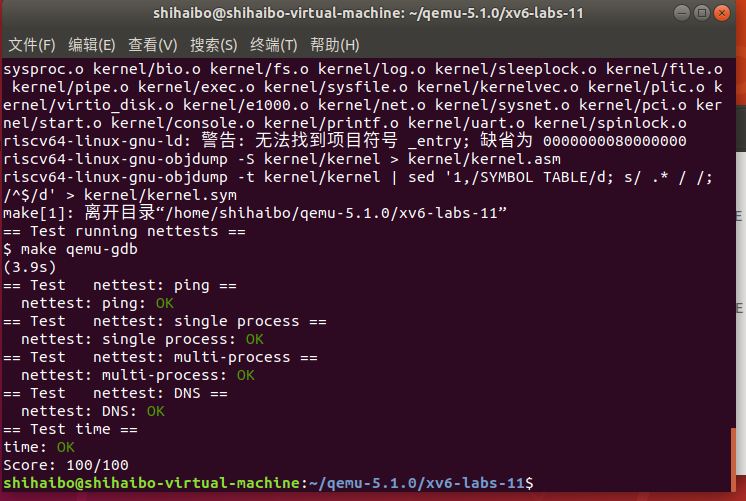
rx\_ring[idx].addr **=** (uint64)rx\_mbufs[idx]**->**head;

regs[E1000\_RDT] **=** idx;

}

}

#### 11.3.2 测试



### 11.4 实验中的问题和解决方法

1.中断处理错误：网络设备产生的中断是处理网络数据包的关键部分。如果中断处理有误，可能导致数据包的丢失或处理错误。检查中断处理程序是否正确地识别和处理中断。

2.数据包接收问题：在实现数据包接收逻辑时，可能会遇到缓冲区管理错误、数据包解析错误或接收队列溢出等问题。确保正确管理和处理接收到的数据包，并避免数据包丢失或内存溢出。

### 11.5 实验心得

1.理解网络设备的初始化和配置：在实验中，我需要了解网络设备的初始化过程，并编写代码来配置设备参数。这使我对如何与硬件进行交互有了更深入的理解。

2.掌握数据包的接收和发送：通过实现网络驱动程序，我学会了如何处理接收和发送网络数据包。这包括封装和解析数据包、构建和解析以太网帧等。这让我更好地理解了数据包在网络中的传输过程。

中断处理和异步操作：在实验中，我需要处理网络设备产生的中断，以及处理异步的数据包接收和发送。这锻炼了我的并发编程和中断处理能力。

3.熟悉网络协议栈的工作原理：通过将网络驱动程序与操作系统内核中的网络协议栈集成，我更深入地了解了网络协议栈的工作原理和网络层次结构。

4.调试和故障排除能力的提升：实现网络驱动程序可能会遇到一些问题和错误，这需要我运用调试工具和技巧来进行故障排除。这让我在代码调试和故障排除方面有了更多的经验。